Rochester Institute of Technology
RIT Digital Institutional Repository

Theses

8-1-2007

Parallelization of the maximum likelihood approach to phylogenetic inference

Janine Garnham

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
Garnham, Janine, "Parallelization of the maximum likelihood approach to phylogenetic inference" (2007). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Parallelization of the Maximum Likelihood Approach to Phylogenetic Inference

by

Janine B. Garnham

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Dr. Muhammad Shaaban
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, NY
August 2007

Approved By:

Dr. Muhammad Shaaban
Primary Advisor – R.I.T. Dept. of Computer Engineering

Dr. Roy Czernikowski
Secondary Advisor – R.I.T. Dept. of Computer Engineering

Dr. Larry Buckley
Secondary Advisor – R.I.T. Dept. of Biological Sciences

Dedication

This work is dedicated to my father, who never did finish his thesis.

Acknowledgements

I would especially like to thank my advisor, Dr. Muhammad Shaaban, along with committee members Dr. Roy Czernikowski and Dr. Larry Buckley for their support and guidance. Your time, effort, and patience are greatly appreciated. I would also like to acknowledge the authors of the fastDNAml program for providing the foundation on which this work was built. Finally, I would like to thank the Research Computing Department at the Rochester Institute of Technology for providing access to the IBM High Performance Computing cluster used to carry out this research.

Abstract

PARALLELIZATION OF THE
MAXIMUM LIKELIHOOD APPROACH
TO PHYLOGENETIC INFERENCE

by Janine B. Garnham

Supervising Professor: Dr. Muhammad Shaaban
Department of Computer Engineering

Phylogenetic inference refers to the reconstruction of evolutionary relationships among various species, usually presented in the form of a tree. DNA sequences are most often used to determine these relationships. The results of phylogenetic inference have many important applications, including protein function determination, drug discovery, disease tracking and forensics. There are several popular computational methods used for phylogenetic inference, among them distance-based (i.e. neighbor joining), maximum parsimony, maximum likelihood, and Bayesian methods. This thesis focuses on the maximum likelihood method, which is regarded as one of the most accurate methods, with its computational demand being the main hindrance to its widespread use. Maximum likelihood is generally considered to be a heuristic method providing a statistical evaluation of the results, where potential tree topologies are judged by how well they predict the observed sequences. While there have been several previous efforts to parallelize the maximum likelihood method, sequential implementations are more widely used in the biological research community. This is due to a lack of confidence in the results produced by the more recent, parallel programs. However, because phylogenetic inference can be extremely computationally intensive, with the number of

possible tree topologies growing exponentially with the number of species, parallelization is necessary to reduce the computation time to a reasonable amount.

A parallel program was developed for phylogenetic inference based on the trusted algorithms of fastDNAml, a sequential program for phylogenetic inference utilizing the maximum likelihood approach. Parallelization is achieved using the popular master/workers scheme, where workers evaluate potential tree topologies in parallel. Three innovative optimizations are employed to alleviate the associated communication bottleneck encountered when using the master/workers technique with large-scale systems and problems. First, message packing reduces the number of messages sent out by the master, along with the associated overheads. Secondly, allowing workers to keep the best trees evaluated reduces the number of messages received by the master, as low-scoring results are discarded by the workers. Finally, multiple masters are utilized to parallelize the responsibilities of what is traditionally a single master process. These last two optimizations led to a dramatic improvement in performance over the unoptimized parallelization under the conditions tested. Message packing, however, demonstrated a slight reduction in performance. Although testing with large-scale systems and problems was not possible, results for all three optimizations suggested likely performance enhancement under such conditions, potentially leading to relief of the bottleneck.

Table of Contents

Dedication iii
Acknowledgements iv
Abstract v
Glossary xii
Chapter 1 Introduction 1
1.1 Goals 2
1.2 Chapter Outline 3
Chapter 2 Background 5
2.1 Phylogenetic Inference 5
2.2 Methods of Phylogenetic Inference 8
2.2.1 Neighbor Joining 8
2.2.2 Maximum Parsimony 9
2.2.3 Maximum Likelihood (ML) 11
2.2.4 Bayesian Methods 12
2.2.5 Disk-Covering Methods 13
2.3 Likelihood Calculation 14
2.3.1 Model of Base Substitution 14
2.3.2 Likelihood of a Tree 17
2.3.3 Branch Length Calculation 20
2.4 Existing Sequential Efforts using ML 20
2.5 Motivation 25
2.5.1 Computational Intensity 25
2.5.2 Communication Bottleneck 27
2.6 Existing Parallel Efforts using ML 29
Chapter 3 fastDNAml 36
3.1 Foundation for Parallelization 37
3.2 Implementation 38
3.2.1 I/O Format & User Options 38
3.2.2 Data Structures 40
3.2.3 Program Flow 44
3.3 Modifications 49
Chapter 4 Parallelization 51
4.1 Approach 51
4.2 Computational Resources 57
4.3 Implementation 59
4.3.1 Message Format 59
4.3.2 Pseudo-code 61
4.4 Difficulties Encountered 68
4.4.1 Achieving Results Identical to fastDNAml 68
4.4.2 Preventing Infinite Rearrangements 70
4.5 Comparison with pfastDNAml 71
Chapter 5 Optimizations to Parallelization 74
5.1 Message Packing 74

5.1.1 Implementation 76
5.1.2 Performance Considerations 82
5.2 Workers Keep Best Trees 83
5.2.1 Implementation 86
5.2.2 Performance Considerations 89
5.3 Multiple Masters 91
5.3.1 Implementation 93
5.3.2 Automatic Selection of Multiple Masters 100
5.3.3 Performance Considerations 101
Chapter 6 The Software 103
6.1 Input Format 103
6.2 Output Format 105
6.3 Program Execution 108
Chapter 7 Results & Analysis 110
7.1 Experimental Conditions 110
7.2 Parallelization without Optimization 111
7.3 Message Packing 115
7.4 Workers Keep Best Trees 122
7.5 Multiple Masters 124
7.5.1 Automatic Selection of Multiple Masters 129
7.6 Combining & Comparing Optimizations 132
7.7 Achieving Results Identical to fastDNAml 136
Chapter 8 Conclusions 138
8.1 Recapitulation 138
8.2 Difficulties 141
8.3 Future Work 143
Bibliography 146

List of Figures

Figure 2.1: Sequence alignment 6
Figure 2.2: Example of a phylogenetic tree 6
Figure 2.3: Simple tree with branch lengths 17
Figure 2.4: Branch swapping 21
Figure 2.5: Parallelization of fastDNAml (pfastDNAml) 31
Figure 2.6: TREE-PUZZLE parallelization 32
Figure 2.7: pIQPNNI parallelization 33
Figure 3.1: Tree node representation 43
Figure 3.2: Sequential program flow for fastDNAml 45
Figure 3.3: Pseudo-code for sequential fastDNAml 48
Figure 4.1: Master/workers sequence diagram (no optimizations) 55
Figure 4.2: Message string format and corresponding topology 60
Figure 4.3: Pseudo-code for the master (no optimizations) 67
Figure 4.4: Pseudo-code for the workers (no optimizations) 68
Figure 5.1: Master/workers sequence diagram (message packing) 76
Figure 5.2: Pseudo-code for the master (message packing) 78
Figure 5.3: Pseudo-code for the workers (message packing) 79
Figure 5.4: Message bundle string format 80
Figure 5.5: Master/workers sequence diagram (workers keep best trees) 85
Figure 5.6: Pseudo-code for the master (workers keep best trees) 88
Figure 5.7: Pseudo-code for the workers (workers keep best trees) 89
Figure 5.8: Organization of processors with multiple masters 93
Figure 5.9: Pseudo-code for the head master (multiple masters) 95
Figure 5.10: Pseudo-code for the extra masters (multiple masters) 97
Figure 5.11: Pseudo-code for the workers (multiple masters) 98
Figure 6.1: Sample input file 104
Figure 6.2: Sample output 108
Figure 7.1: Execution time results (no optimizations) 112
Figure 7.2: Speedup results (no optimizations) 113
Figure 7.3: Execution time results (message packing with 10 trees per message) 116
Figure 7.4: Speedup results (message packing with 10 trees per message) 117
Figure 7.5: Execution time results (message packing with 5 trees per message) 117
Figure 7.6: Speedup results (message packing with 5 trees per message) 118
Figure 7.7: Execution time results (message packing with 3 trees per message) 118
Figure 7.8: Speedup results (message packing with 3 trees per message) 119
Figure 7.9: Execution time results (comparison between message sizes) 121
Figure 7.10: Execution time results (workers keep best trees) 122
Figure 7.11: Speedup results (workers keep best trees) 122
Figure 7.12: Execution time results (multiple masters with 3 extra masters) 125
Figure 7.13: Execution time results (multiple masters with 5 extra masters) 125
Figure 7.14: Execution time results (multiple masters with 8 extra masters) 126
Figure 7.15: Execution time results (multiple masters with 10 extra masters) 126
Figure 7.16: Speedup results (multiple masters with 3 extra masters) 127
Figure 7.17: Speedup results (multiple masters with 5 extra masters) 128

Figure 7.18: Speedup results (multiple masters with 8 extra masters)................................128
Figure 7.19: Speedup results (multiple masters with 10 extra masters)..............................129
Figure 7.20: Execution time results (comparison between numbers of extra masters) 130
Figure 7.21: Execution time results (automatic selection of extra masters) 131
Figure 7.22: Speedup results (automatic selection of extra masters) 132
Figure 7.23: Execution time results (multiple masters, workers keep best trees)...............133
Figure 7.24: Speedup results (multiple masters, workers keep best trees)..........................134
Figure 7.25: Execution time results (3 optimizations combined)..134
Figure 7.26: Speedup results (3 optimizations combined) 135
Figure 7.27: Speedup comparison for individual and combined optimizations 136

List of Tables

Table 3.1: fastDNAml user options..39
Table 3.2: Data structures of fastDNAml..41

Glossary

Bottleneck	Situation limiting the full potential of parallelization.
Branch Length	Reflects evolutionary distance between two species connected by a branch in a phylogenetic tree, and is proportional to time.
Branch Swapping (Rearrangements)	Transformation of an existing topology into new topologies, where branches are cut and reinserted at different locations.
Cluster	Group of autonomous computers that work together in parallel, generally communicating via message-passing over a network.
DNA Sequence	Specific ordering of the four types of base pairs, or nucleotides, forming DNA, the molecule responsible for heredity. Represented as a text string containing letters A, G, C, and T, and serves as input for phylogenetic inference.
Heuristics	Methods used to reduce the number of potential topologies to evaluate by focusing on a promising subgroup.
Likelihood	Probability of a particular tree producing the observed input sequences. Used as the optimality criterion to compare potential topologies with the maximum likelihood method of phylogenetic inference.
Master/Workers Scheme	Approach to parallelization where a single master processor coordinates division of the workload over multiple worker processors operating in parallel.
MPI	Message Passing Interface, a popular library specification used to achieve parallelization in message-passing environments such as clusters.
Phylogenetic Inference	Reconstruction of evolutionary relationships among various species, usually presented in the form of a tree.
Taxon	For purposes of this thesis, refers to a species input for phylogenetic inference with a corresponding DNA sequence, and appears at a leaf, or tip, of the phylogenetic tree (plural taxa).
Topology	Structural configuration of a phylogenetic tree specifying the relative locations of species. Used interchangeably with ‘tree’ for purposes of this thesis.

Chapter 1 Introduction

Historically the physical attributes of organisms have been used in phylogenetic inference to create evolutionary trees. With the advent of computer technology together with the continuously increasing amount of DNA data available, researchers today most commonly use computer programs for phylogenetic inference. This has lead to more accurate results, which are achieved in a shorter amount of time. However, even with computer processing there is still no way to ascertain that a derived phylogenetic tree is accurate, as evolution occurred in the past and is thus not observable. Several computational methods of phylogenetic inference exist with varying levels of regarded accuracy. Simplified models and inaccurate assumptions are used by all methods to make program development tractable. In general, there is a tradeoff between accuracy and speed. A popular computational method employed is maximum likelihood, which is more accurate and more computationally intensive than other popular phylogenetic inference methods such as neighbor joining and maximum parsimony [14]. Maximum likelihood is a standard method used in statistics and uses an optimality criterion for judging potential tree topologies based on how well the tree predicts the observed input sequences. The tree calculated to have the highest probability, or likelihood, of producing the sequences is considered to be the most accurate tree. The use of maximum likelihood in phylogenetic inference was popularized by Felsenstein in 1981 with the program DNAml [8]. An updated version of DNAml is currently available as part of the PHYLIP package [10]. Many of the maximum likelihood algorithms developed later were based on the DNAml algorithm, including fastDNAml, which was published in 1994 by Olsen, Matsuda, Hagstrom and Overbeek and offers enhanced performance over DNAml [23]. fastDNAml has since been used as a foundation

for more many advanced algorithms. Maximum likelihood program development continues to be an active area of research. Much of the recent research has focused on parallelization of phylogenetic inference programs as parallel computing systems have become more available. The aim of parallelization is to reduce execution time by dividing the workload over many processors that execute in parallel. Typically a master/workers scheme is used where a single master, or head, processor is responsible for the main program execution, coordinating the division of the workload over multiple slaves and deciphering the results. However, with the trend toward larger, more powerful parallel computing systems and larger datasets a communication bottleneck has arisen [49]. On modern systems the number of workers can easily reach thousands and a single master will struggle to keep up with the communication demands of so many workers, compromising the full potential of the parallelization. Relief of this bottleneck is the focus of this thesis.

1.1 Goals

For this thesis, parallel phylogenetic inference software using the maximum likelihood approach and the master/workers scheme was developed. The popular sequential fastDNAml software served as the starting point for parallelization due to its relative simplicity, availability, and acceptance in the research community. The highlight of the research presented is the relief of the communication bottleneck in three different ways, together with an analysis of the bottleneck. The software developed provides a parallelization of a program whose results are trusted in the research community, with applicability to large systems and problems. In addition it is the hope that the ideas presented will aid the research community by providing solutions that can be applied to existing parallel phylogenetic inference programs that utilize more advanced and complex

methods that are restricted to small-scale systems and problems due to the bottleneck. For this reason the software implementation will be made freely available to the public as open-source code. The program was developed for a message-passing environment using the MPICH library [22] together with the C programming language. The input and output formats follow those used by the widely-popular PHYLIP package [10]. It is the intention that the software will be incorporated into a package along with other parallel programs using different methods which are currently under development in the research group.

1.2 Chapter Outline

The material presented in this document is laid out as follows. Chapter 2 provides the background information necessary to appreciate this thesis and put it into perspective. This includes an introduction to phylogenetic inference and the maximum likelihood method, a description of previous related efforts, and an overview of the research carried out and how it contributes to the field. Chapter 3 introduces the sequential fastDNAml program with an explanation of why it was selected as a foundation for this work, details on its implementation, and how it was modified in preparation for parallelization. The parallelization of fastDNAml, before inclusion of the optimizations for bottleneck relief, is provided in Chapter 4. The basic approach is given along with a description of the implementation and computational resources utilized. This description of the basic parallelization strategy then lays the foundation for Chapter 5, which presents the core of the research. Here, the three modifications to the master/workers scheme are described in detail, including their implementation and how they lead to a reduction of the associated communication bottleneck. Chapter 6 covers usage of the resulting software, specifically how each of the three optimizations is selected by the user, along with the expected input

and output formats. The results of the research are presented in Chapter 7, with the three optimizations analyzed individually and in combination. The document concludes with Chapter 8, which provides a summary of the work highlighting the accomplishments, followed by a description of the difficulties encountered and suggestions for future improvements.

Chapter 2 Background

In order to put this work into perspective, an appropriate background is first needed and is the focus of this chapter. An introduction to phylogenetic inference is provided, followed by a discussion of the various methods used, as well as why the maximum likelihood method, specifically, was selected. Details of how phylogenetic inference is accomplished through maximum likelihood are also covered. Finally, the motivations for this work are described, including the limitations of previous parallelization efforts, which this work aims to overcome.

2.1 Phylogenetic Inference

Phylogenetic inference is most often accomplished through the use of a computer program that takes aligned DNA sequences, one for each organism, as input and produces its estimate of the most accurate tree as output. DNA, or deoxyribonucleic acid, consists of strings of four nucleotides, or bases, adenine (A), guanine (G), cytosine (C), and thymine (T), the sequence of which code for proteins. Proteins are ultimately responsible for the attributes making each species unique. As time passes, DNA mutates, forming slightly different versions of the same protein, and thus different species, creating the phenomena termed evolution. One common type of mutation is the substitution of one base pair for another. This is demonstrated in Figure 2.1, where a portion of the DNA sequence for a particular protein is shown aligned for five species. The mutations are displayed in color.

```
Ra 1    CCCCAGGGTGGTGGCTGGGGGCAG
Rb 2    CCTCATGGTGGTGGCTGGGGGCAA
Rc 3    CCCCATGGTGGCGGCTGGGGACAG
Rd 4    CCCCATGGTGGCGGATGGGGACAG
Re 5    CCTCATGGTGGCGGCTGGGGTCAA
```

Figure 2.1: Sequence alignment displaying mutations for five species [25].

It is generally assumed that the greater the evolutionary distance between two species, the larger the number of differences between their respective DNA sequences coding for particular proteins. This idea is used in phylogenetic inference, where DNA sequences from the genes of various taxa (i.e. species) are analyzed to determine their relative evolutionary distances. The results are then displayed in the form of a phylogenetic tree, such as in Figure 2.2.

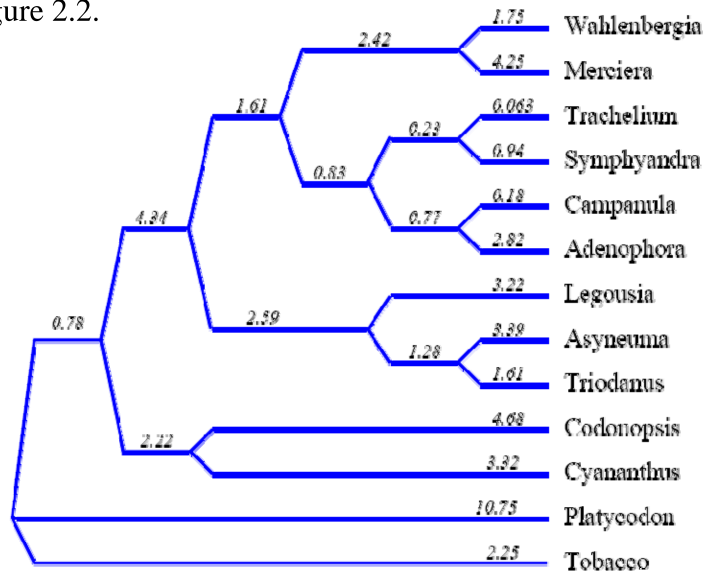


Figure 2.2: Example of a phylogenetic tree for the *Campanulaceae* (Bluebell flower) family [3].

Note that the 13 species presented in this example are located at the tips of the tree branches. The interior nodes represent hypothetical common ancestors. Rooted, bifurcating trees such as this example are most often used in phylogenetic inference, although unrooted

trees are commonly used internal to phylogenetic inference programs [8]. The labeled branch lengths reflect evolutionary distance, which is proportional to time and in turn recency of common descent. For instance, according to the tree, Asyneuma is more closely related (shares a more recent common ancestor) with Triodanus than with Platycodon. The branch lengths are derived from the differences in the respective DNA sequences. Unfortunately, the assumption that evolutionary distance is directly proportional to differences in DNA sequences does not always hold [14], making phylogenetic inference significantly more complex when estimating relationships. The term 'inference' is used because the evolutionary relationships must be inferred from the available data, with the true relationships unknown. This can make judging the accuracy of a resulting tree difficult.

Phylogenetic inference utilizes models of sequence evolution that describe the process of substitution of nucleotides along the branches of a phylogenetic tree. Substitutions at each site in the DNA sequences are generally considered to be independent. These models are used to derive the probability of substitution of any nucleotide for any other nucleotide at each of the sequence sites over a period of evolutionary time. Empirical and parametric models of sequence evolution exist, with empirical models being less accurate but computationally simpler than parametric models. Empirical models incorporate fixed parameters whose values are assigned based on previous analysis of large volumes of sequence data rather than on the data to be analyzed. Parametric models, on the other hand, base parameter values on the sequences under analysis, with the values updated as the analysis progresses. The three most common types of parameters used are base frequency, base exchangeability, and rate heterogeneity. Base frequency parameters represent the

relative occurrences of the four bases over all sites and sequences in a tree, while base exchangeability parameters describe the relative probabilities that a particular base will be substituted for another. Some models take into account similarities in the chemical structures between the bases in the derivation of the exchangeability parameters. The bases are classified into two groups based on their structure, where adenine (A) and guanine (G) are both purines, and cytosine (C) and thymine (T) are both pyrimidines. On a substitution, a purine is more likely to replace a purine, and a pyrimidine is more likely to replace a pyrimidine. The term transition is used to describe either of these substitutions. When a purine is substituted by a pyrimidine, or vice versa, a transversion occurs. The relative rate of transitions to transversions is often described as a ratio, which can be factored into the base exchangeability parameter calculations. Rate heterogeneity parameters make use of a gamma distribution to derive the rates of mutation at each site in the DNA sequences. Research into creating more accurate and complex models of sequence evolution is ongoing [46].

2.2 Methods of Phylogenetic Inference

2.2.1 Neighbor Joining

Neighbor Joining [40] is a fast, distance-based method for phylogenetic inference introduced by Saitou and Nei in 1987 [29]. Tree topology and branch lengths are generated from a distance matrix comprised of the estimated evolutionary distances between each pair of input sequences. Evolutionary distance represents the amount of evolution, or nucleotide substitutions, between two sequences since diverging from a common ancestor, and is generally determined using an empirical model of sequence evolution. Scores are

assigned to each pair of sequences based on the matrix. The algorithm initially joins the pair with the lowest score, forming a subtree. The resulting root formed by this pair then replaces these two sequences in the matrix, and the distances in the matrix are recalculated. This joining process continues until only two entries remain in the matrix, which are then joined together to form the final unrooted binary tree.

The main disadvantage of the neighbor joining method is the loss in accuracy due to the conversion of sequences into a distance matrix. This is because the observable differences among sequences are not always proportional to evolutionary distance, especially for divergent sequences [14]. This method is used mainly because of its speed, and the results are often used as starting trees for more complex methods. No major parallel implementations of the neighbor joining method exist, possibly due to the lack of computational intensity.

2.2.2 Maximum Parsimony

Maximum parsimony is a character-based method, rather than distance-based, where the sequence characters themselves are used in determining the optimal tree. This gives maximum parsimony increased accuracy over neighbor-joining as the information lost on conversion to distances is preserved. Maximum parsimony uses an optimality criterion to evaluate potential topologies, where the best tree is that requiring the minimum number of mutations to achieve the observed sequences. The method assumes that the preferred path of evolution from the ancestral node to the input sequences is that with the fewest number of changes in the sequences. The name maximum parsimony indicates that the most frugal solution is favored, that is, the tree yielding the minimum number of mutations to arrive at

the data. Maximum parsimony differs from the other methods of phylogenetic inference in that it lacks a specific model of sequence evolution. However, it is similar to maximum likelihood in that many different search algorithms, exact or heuristic, exist to find potential topologies, which are ultimately compared using an optimality criterion. There are also various scoring strategies that have been employed in maximum parsimony, one of the most popular being Fitch's strategy [2].

To determine a score using Fitch's strategy, an arbitrary root of the tree is set as the calculation root. Treating each base, or sequence site, independently and moving from the leaves toward the calculation root, character sets are derived for each node as the tree is simultaneously scored. The character set for each of the leaf nodes contains the character in that sequence for the site analyzed. A parent node's character set is then found from the intersection of its children's character sets. If two children share no characters in common, then the parent's set becomes the union of the children's sets. When this occurs the score of the tree increases by one, making it less parsimonious. The total score, or number of mutations, for the tree is thus known once the calculation root is reached. The lowest scoring tree overall is selected as the best tree.

As with neighbor joining, maximum parsimony is used more for its speed than for its accuracy [14]. Only one mutational mapping is considered, being that with the fewest mutations, while many other mappings are possible. These other trees could potentially have more mutational pathways that explain the data than the tree with the fewest mutations, and thus should be considered as better alternatives. An additional drawback is that maximum parsimony performs poorly with divergent sequences. This is because

convergence, which increases with more divergent sequences, is not accounted for as a cause of similarity between sequences. Several parallel implementations of the maximum parsimony algorithm exist, such as ExactMP [2] and an effort by C. Howard and M. Shaaban at the Rochester Institute of Technology [15].

2.2.3 Maximum Likelihood (ML)

The popular character-based method of maximum likelihood was the method of choice for this work as its greater computational intensity over neighbor-joining and maximum parsimony makes it well-suited for parallelization. As with maximum parsimony, an optimality criterion is used to judge all potential topologies that can be generated from the input sequences. The topology with the highest calculated likelihood is considered to be the best tree. The likelihood of a tree is the probability of obtaining the input sequences, found at the tips, given the specific tree topology and branch lengths. Since the likelihood is maximized over all potential tree topologies rather than over the sequence data, the sum of the likelihoods will not equal one. A tree's likelihood is a reflection of how well the tree topology predicts the sequences, not the probability that the topology is accurate. Details on calculating the likelihood of a tree will be provided in a later section.

While it is possible to evaluate all potential tree topologies using maximum likelihood, the time and computational requirements due to the complexity of the maximum likelihood calculations make it impractical for large numbers of species. Therefore, maximum likelihood is generally considered to be a heuristic method rather than an exact method, as heuristics are almost always utilized to reduce the search space. Unfortunately, when using

heuristics it is not guaranteed that the best tree will be found as not all topologies are evaluated and local optima often hide the global optimum.

2.2.4 Bayesian Methods

Bayesian inference [48] [14] [17] is a recent, nontraditional method derived from the field of Bayesian statistics and based on the maximum likelihood method. Bayesian inference is sometimes treated as a maximum likelihood method, but it is generally considered to be a separate method. Bayesian inference judges trees based on their posterior probabilities, which serve as the output of the analysis. According to Bayes' theorem, posterior probability is proportional to both the likelihood and prior probability of the tree. The prior probability is the probability of the tree without considering the input sequences, and can be based on first principles, general knowledge, or past experiments. Posterior probabilities reflect the probability that a tree is correct and sum to one over all results, unlike likelihood values. The tree with the maximum posterior probability is generally taken to be the best tree.

To determine posterior probabilities, integration over all sequence model parameter values in combination with all possible branch lengths for each tree must be performed. Often the numerical Markov chain Monte Carlo (MCMC) algorithm is used to find approximations of the posterior probabilities. This algorithm forms a chain of steps through parameter space, where each new step represents a perturbation of the current tree. Perturbations are accepted when resulting in an increased relative posterior probability density, otherwise they are rejected with a certain probability. The relative frequency that locations in

parameter space are visited correspond to the posterior probability of the tree and sequence evolution model parameters represented by that region of parameter space.

One of the major difficulties faced when using MCMC is determining when the chain has progressed far enough to provide reliable estimates of the posterior probabilities. Another issue with Bayesian inference is the subjectivity of the prior probabilities. Further, Bayesian inference is far more computationally complex than maximum likelihood, which hinders its widespread use. This is partially due to the use of significantly more complex models of sequence evolution over maximum likelihood. However, even with these weaknesses, Bayesian methods have been shown to outperform the other methods of phylogenetic inference [48]. Parallelizations of MCMC-based Bayesian inference have been developed [1] [27], mostly based on the popular sequential program MrBayes [18].

2.2.5 Disk-Covering Methods

The family of Disk-Covering Methods (DCM) introduced by Warnow et al. [19] [28] are not traditional phylogenetic inference methods like neighbor joining, maximum parsimony, or maximum likelihood. Instead they are used along with phylogenetic inference methods, acting as a performance booster. Performance is enhanced by using a divide-and-conquer approach to reduce the problem size. The input data are divided into subgroups with overlapping elements. A specific phylogenetic inference method then acts on each of the subgroups, resulting in many subtrees. In general, phylogenetic inference methods yield more accurate results with smaller datasets, and the computation time is reduced. The subtrees are merged into one supertree that serves as the resulting phylogenetic tree for the program. The Strict Consensus Subtree Merger is one of several merging techniques

available, which takes advantage of the placement of the data elements shared between subtrees to perform the merge. Variants such as DCM1, DCM2, and DCM3 [28] exist, differing in their strategies on division of the dataset. DCM is well-suited to parallelization as there is a distinct division of the workload, where generation of each subtree can be accomplished in parallel on separate processors.

2.3 Likelihood Calculation

This thesis work is based on the maximum likelihood method. The core of this method is the calculation of the likelihood value of a given tree. A model of base substitution is needed to determine the probabilities of base change, which is used along with the branch lengths and sequence data to determine the likelihood. The end result of the method is the tree with maximum likelihood of all trees evaluated.

2.3.1 Model of Base Substitution

Calculation of the likelihood of a tree involves not only the topology and branch lengths, but a model of base substitution is needed as well. For the fastDNAml program, and thus this thesis work, the F84 model is used [11] [20] [9]. This model has five parameters, allowing for unequal equilibrium base frequencies and differing rates of transition versus transversion substitutions. The equilibrium base frequencies refer to the relative amounts of each of the 4 bases, A, G, C and T, which should be conserved over the period of evolution. Representations of π_A, π_G, π_C, and π_T are used and their values can be determined from the input sequence data. The relative rates of transition vs. transversion are generally input in the form of a ratio. The model is based on the Markov process where the future state, or

base value, of a site is dependent only on the value of its current base, irrespective of the history of base changes [8] [42].

This model uses two types of events that may take place at a sequence site. These events are not accurate reflections of the true process of base substitution, and are simply used to model the process. An event of type I is the random drawing of a new base (to replace the current base) from a pool of either purines or pyrimidines, dependent on whether the current base is a purine or pyrimidine, respectively. This results in either no change or a transition. The ratio of the two bases in each pool is based on their equilibrium base frequencies. If the base is originally a purine (A or G), the probability of pulling an A from the pool is $\pi_A/(\pi_A + \pi_G)$, and the probability of pulling a G is $\pi_G/(\pi_A + \pi_G)$. Similar probabilities are calculated for pulling a C or T when the base is originally a pyrimidine. A rate of α is given to event type I, with no difference whether purines or pyrimidines are involved. Thus, the probability of an event of type I occurring along a branch representing time t is $(1-e^{-\alpha t})$, and the probability of no type I event occurring during that time is $e^{-\alpha t}$.

Event type II involves the pulling of a base at random from a pool containing all four bases, potentially causing no change, a transition, or a transversion. The bases are in the pool at same relative concentrations as their equilibrium base frequencies, and the probability of pulling each of the 4 bases is thus equal to its equilibrium base frequency. A rate of β is given to the occurrence of event type II, and therefore the probability of event II occurring during time t is $(1-e^{-\beta t})$, and that of it not occurring is $e^{-\beta t}$.

Before calculating the likelihood of a given tree one first needs to determine the transition probabilities, also known as base exchangeability parameters. These are the probabilities of

change from a given base type to any of the four base types. There are 16 possible transitions, including a 'change' to the same base type, for example from A to A, which is unobservable. Note here that the term 'transition' is being used in a sense different from earlier when discussing its relation to transversions. A transition from base i to new base j, where i and j are A,G,C, or T, has a transition probability represented by P_{ij}. Reversibility is a property of the model of base substitution used such that $P_{ij} = P_{ji}$, with the probability remaining the same regardless of which base, i or j, was the original vs. new base.

Simplification of the transition probability calculations can be achieved with the realization that when at least one event of type II occurs during time t, pulling any of the four bases at random, then all other events during that time, either type I or II, do not affect the result. Recall that the probability of at least one event of type II occurring during time t is $(1-e^{-\beta t})$, and the probability of getting base j on that event is $\pi_{j.}$ If there are no events of type II during time t, but at least one of type I, regardless of how many, then the probability of getting base j during time t is based on a single type I event. For instance the probability of getting an A when the original base is a purine is simply $\pi_A/(\pi_A + \pi_G)$. The probability that there is an event of type I and none of type II is $(e^{-\beta t})(1-e^{-\alpha t})$. The probability that no events occur during time t, and thus no base change, also needs to be considered, which is equal to $e^{-(\alpha+\beta)t}$. A general expression can be written for all 16 transition probabilities by utilizing the Kronecker delta function, δ_{ij}, and its Watson-Kronecker equivalent, $\in_{ij}$. The Kronecker delta function is equal to one when bases i and j are identical and zero otherwise, and the Watson-Kronecker equivalent equals one when i and j are both purines or both pyrimidines, and zero otherwise. The general expression [11] is

$$P_{ij} = e^{-(\alpha+\beta)t}\delta_{ij} + e^{-\beta t}\left(1-e^{-\alpha t}\right)\times\left(\frac{\pi_j}{\sum_k \pi_k \in_{jk}}\right)\in_{ij} + \left(1-e^{-\beta t}\right)\pi_j \qquad \text{(Eqn. 1)}$$

Note that the summation in the denominator of the second term is equivalent to either $\pi_A+\pi_G$ or $\pi_C+\pi_T$ if j is a purine or pyrimidine, respectively. The values for α, β, and t are determined from many values including the transition-transversion ratio, site-specific rates of evolution, equilibrium base frequencies, and the branch lengths of the associated nodes.

2.3.2 Likelihood of a Tree

With the transition probabilities determined above, the likelihood calculation for a full tree can be now be described [8] [11]. Refer to the simple tree in Figure 2.3 below as an aid.

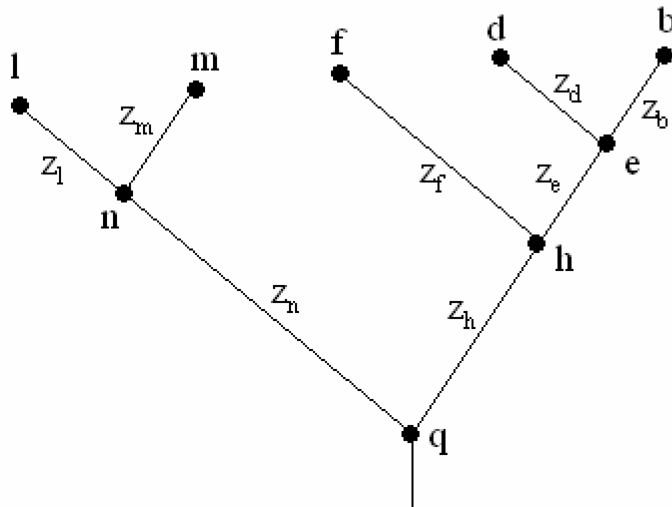


Figure 2.3: Simple tree with branch lengths (z).

Recall that when calculating the likelihood of a tree it is in fact the probability of the specific input sequences arising on the tips of the tree that is being calculated. To make the calculation feasible, the somewhat invalid assumption is made that each sequence site, or

column in the alignment of sequences, evolves independently of the other sites, or columns. This allows the likelihood contribution from each site to be calculated separately. Once the contributions are calculated, the product of those values is then taken over all sites to arrive at the overall likelihood value for the entire tree. Therefore the likelihood calculation for only a single site will be provided. The calculation involves a postorder tree traversal, where the conditional likelihoods of each tree node are calculated, starting at the tip nodes and moving down toward the root of the tree. For the model used, the root may be placed anywhere on the tree without affecting the resulting likelihood [8].

Each tree node has four conditional likelihoods, one for each potential base state. Node e, for instance, has conditional likelihoods denoted by L_e, which represent the likelihood of the tree contributed by node e and all nodes above it, toward the tree tips, for the particular base state of e at the site under consideration. Calculation of the conditional likelihoods for the tip nodes is straightforward as the actual base states are known from the sequence data. The probability of the observed base occurring is one, as is the conditional likelihood for that base. The probabilities and conditional likelihoods for the remaining three bases are thus zero. No nodes exist above the tips and consequently the transition probabilities do not come into play. Calculation of the conditional likelihoods for the internal nodes proves more complicated, however, as the true base states are unknown. The probability of finding each of the potential base states at a given internal node needs to be determined using the transition probabilities from each of the potential base states of the two descendent nodes along with the probabilities of finding each of the four bases states at the descendent nodes, i.e. their conditional likelihoods. This is exhibited in Equation 2 below for node h in Figure 2.3 and its descendent nodes e and f.

$$L_h = \left(\sum_f P_{hf}(z_f) L_f \right) \left(\sum_e P_{he}(z_e) L_e \right) \qquad \text{(Eqn. 2)}$$

There are four such equations for the conditional likelihoods associated with the four potential base states at node h. The transition probabilities, between node h and its descendent node e for example, are denoted as $P_{he}(z_e)$. The z_e signifies that the transition probability is a function of the particular branch length under consideration. A summation is taken over the four potential base states of both descendent nodes e and f.

Similar calculations are carried out for every internal node of the tree, moving from the tips toward root node q, with each node's calculation using the conditional likelihoods calculated previously for the associated descendent nodes. The four conditional likelihoods of node q are calculated in the same way using conditional likelihoods of descendent nodes h and n. Each of these conditional likelihoods for node q is then combined into a product with the associated equilibrium base frequency, which represents the probability that the state of node q is that particular base. A summation is taken over the four potential base states at q to obtain the likelihood of the tree for the site of interest, as shown in Equation 3.

$$L = \sum_q \pi_q L_q \qquad \text{(Eqn. 3)}$$

Once the likelihood contributed by every site is determined, the overall likelihood of the tree is determined by taking the product of the likelihoods over all sites. Because the values for the likelihoods are generally too small to be accurately represented as standard floating-point values, the natural logarithm of the likelihood is generally used.

2.3.3 Branch Length Calculation

Finding the maximum likelihood tree involves not only a determination of the topology, but the optimized branch lengths for the tree need to be computed as well. The branch lengths are optimized in such a way that they maximize the likelihood value for a particular tree. The fastDNAml program, and thus this thesis work, takes the following approach [23] [11]. Each branch length is optimized individually. When optimizing a particular branch length, the location of the root of the tree is taken to be in that branch. The conditional likelihoods of the tree nodes are calculated and values are obtained for the two nodes at the ends of the branch to be optimized. The conditional likelihoods for the root, and thus the likelihood for the tree, are then functions of that branch length. The branch length is iterated using the Newton-Raphson method [11] to find the value that results in the maximum likelihood value for the tree. This process is repeated for all branches in the tree. Several passes are made over the tree until the branch lengths stop changing significantly or a maximum number of passes has been reached. When the branch length optimization is complete, calculation of the final conditional likelihoods is also complete and computing the likelihood of the whole tree becomes straightforward.

2.4 Existing Sequential Efforts using ML

A typical sequential algorithm using the maximum likelihood method is performed as follows [13]. An initial unrooted, bifurcating tree is created from three taxa, of which there is only one unique configuration. A fourth taxon is added to this initial tree. Three distinct new trees are formed, with the forth taxon added at each of the three possible branch locations. The branch lengths in each of these new trees are optimized, the likelihood is calculated, and the best tree, that with the maximum likelihood, is kept. This

process of tree building, termed stepwise addition, continues with the addition of the next taxon to the current best tree, forming (2i-5) new topologies containing i taxa, which are then evaluated. After each addition of a new taxon, topological rearrangement is often performed on the calculated best tree forming a number of new topologies. This is also termed branch swapping as a branch is cut, removing a subtree from the tree, and then reinserted into another existing branch. This is demonstrated in Figure 2.4 below, where the subtree containing tips A and B is removed and reinserted into the branch between G and its most recent ancestor shared with F.

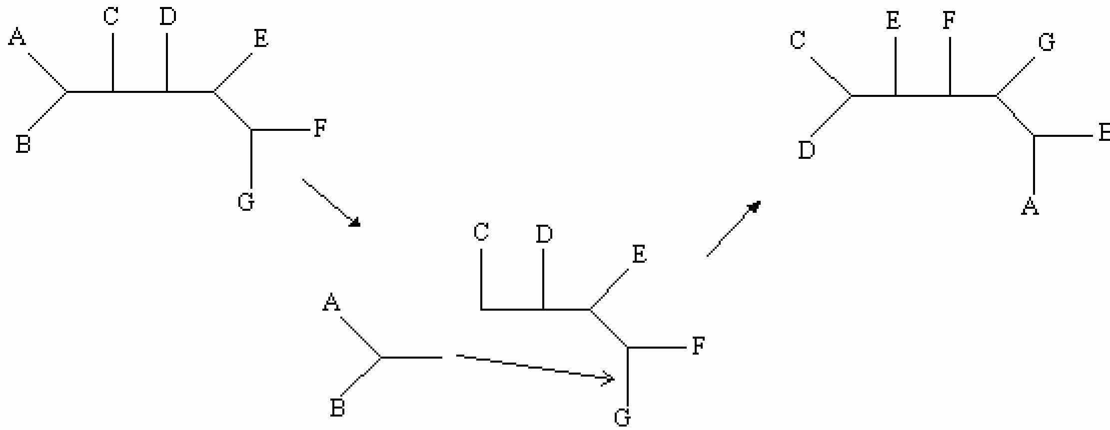


Figure 2.4: Generation of a new topology via branch swapping [42].

The branch lengths and likelihood of each resulting rearrangement are evaluated and the best-rearranged tree is kept, to which additional taxa are added. Once all taxa have been added to the tree, more intense topological rearrangements are carried out until no further rearrangements result in an improved likelihood. This final rearranged tree is considered to be the most accurate topology and is output by the program, along with its associated likelihood value. This hill-climbing algorithm is implemented by several popular maximum

likelihood programs including those in the PAUP* package [41], DNAml in the PHYLIP package [10], and fastDNAml [23]. fastDNAml serves as the sequential foundation for the parallelization performed as part of this thesis work, and is described in more detail in the next chapter.

The PHYML program [13] uses a different algorithmic approach to the maximum likelihood method where an initial tree containing all taxa is generated using a fast distance-based method of phylogenetic inference such as neighbor joining. Simultaneous rearrangements and branch length optimizations are then performed on this starting tree with each iteration of the program. This hill-climbing algorithm ensures that the likelihood will only improve until an optimum is reached.

The maximum likelihood method has also been used with the quartet puzzling algorithm PUZZLE [39] consisting of three steps. In the first step, the *ML step*, multiple maximum likelihood trees are constructed from quartets, or sets of four input sequences. The second step is referred to the *puzzling step* where sequences are added one at a time to the quartet trees forming an intermediate tree. Multiple intermediate trees result from repeating the puzzling step with varying orders of input sequences. The *consensus step* is the third step where the majority rule consensus tree is taken as the output of the algorithm.

The IQPNNI program [45] uses a modified, more efficient version of quartet puzzling called the important quartet puzzling algorithm (IQP). Trees are initially created using a fast neighbor joining method and then subject to nearest neighbor interchanges (NNIs), a particular form of branch swapping. The NNIs continue until a maximum likelihood-based local optimum is found. This tree is then further optimized by randomly deleting leaves and

then randomly reinserting them according to the important quartet puzzling algorithm. NNIs are again performed until a local optimum is reached, with this improved tree then subject to further random deletions and insertions, followed by NNIs. This process is repeated until either a specified number of iterations have taken place or a stopping criterion is satisfied.

GARLI [51] [50], or Genetic Algorithm for Rapid Likelihood Inference, is one of the most recent and best performing sequential maximum likelihood programs available. As the name suggests, GARLI uses genetic algorithms to achieve the maximum likelihood solution by simultaneously modifying tree topology, branch lengths and sequence evolution model parameters. Each individual of the evolving population represents a topology along with the corresponding branch lengths and model parameters. A score is given to each individual based on its likelihood. Random mutations are applied to the individuals at each generation. Mutations may be topological mutations, specifically subtree rearrangements followed by branch length optimizations, or branch length or model parameter mutations, where the values are multiplied by a gamma-distributed variable. The individuals with the best scores following the mutations serve as parents for the next generation of individuals, in proportion to their scores. As this process repeats, the population evolves toward higher likelihood solutions.

Another top performer is RAxML, or the Randomized Axelerated Maximum Likelihood program [32] [36] [34]. RAxML initially uses a starting tree containing all taxa generated using the fast maximum parsimony method. A technique known as lazy subtree rearrangements is then used to analyze alternative topologies. As opposed to fastDNAml,

where all branch lengths in each new topology are optimized following a subtree move, RAxML initially only optimizes the branch lengths neighboring the insertion point. The branch lengths of the topologies with the resulting top likelihoods are then fully optimized, the likelihoods are recalculated, and the maximum likelihood tree is selected. This pre-scoring allows for faster evaluation of a larger number of potential topologies. These lazy subtree rearrangements are utilized by either of the two different search algorithms offered by RAxML. The user has the option of selecting between the hill-climbing procedure, which continues with the rearrangements until the likelihood ceases to improve, and the simulated annealing procedure, which, although slower than hill-climbing, is able to avoid local maxima.

RAxML-V, as described above, is currently considered to be the standard version of the many phylogenetic inference programs based on the maximum likelihood approach developed by Stamatakis [34]. The most recent program, published in November, 2006, is RAxML-VI-HPC, a specialized version of RAxML for large datasets containing over 1,000 taxa. For certain large real datasets, RAxML-VI-HPC was shown to outperform other top sequential likelihood programs such as GARLI, PHYML, IQPNNI, MrBayes [18] and RAxML-V, generating better results and using less main memory in a shorter amount of time. These other programs generally require less than 24 hours to generate a tree for 1,000 taxa.

These popular programs illustrate the two basic approaches used to finding the maximum likelihood solution from a set of input sequences. A tree is either built through stepwise addition of sequences as the algorithm searches for the best tree, or the program performs

manipulations on a preconstructed tree containing all taxa. Regardless of the approach, all programs discussed here utilize heuristics in order to reduce the search space of potential topologies.

The programs described in this section were designed for execution on sequential computing platforms. Programs using the maximum likelihood approach have been designed for parallel execution as well, most of which are based on existing sequential algorithms. Details of the parallelizations will be discussed in a later section.

2.5 Motivation

2.5.1 Computational Intensity

Phylogenetic inference is an incredibly computationally intensive problem. The number of possible unrooted, bifurcating tree topologies grows exponentially with the number of organisms n, based on the formula [7]

$$\frac{(2n-5)!}{2^{(n-3)}(n-3)!} \qquad \text{(Eqn. 4)}$$

Thus, for 50 organisms there are 2.8 x 10^{74} possible tree topologies, and for 100 organisms there are 1.7 x 10^{182} possible topologies, a value greater than the number of atoms in the universe [33]. Finding the most accurate tree among these possible topologies is considered to be NP complete [4] [5]. One of the grand challenges in bioinformatics is the construction of the tree-of-life containing all organisms on Earth [37]. Over the last decade, the amount of genetic sequence data available to researchers has increased rapidly, thus raising the need for phylogenetic inference for a larger number of sequences. As of 2006 there were

over 61 million sequences available in the genetic sequence database GenBank [12], and this number is continuously increasing. Trees generated through phylogenetic inference may typically contain thousands of sequences, each several thousand bases long.

Many factors affect the amount of computation in addition to the size of the dataset, including the method of phylogenetic inference used. The maximum likelihood method is one of the most computationally intensive methods available, which is its main drawback [38] [14]. Regardless, it remains very popular as it is provides more accurate and consistent results as compared to other less intensive methods such as maximum parsimony and neighbor joining [14] [49]. Studies indicate that accuracy equal to or greater than 99.99% may be necessary for the results of phylogenetic inference to be considered useful, thus validating the need for complex computational methods [47]. Further increasing the demand for computational resources are the multiple runs often performed with randomizations of the input data or bootstrapping, used to gain an even better estimate of the true tree.

The proposed solution to this computational burden is parallelization, specifically of the maximum likelihood method. This will enable more data to be analyzed faster, and provide a more accurate estimate of the best tree in a reduced amount of time. While previous efforts have been made to parallelize maximum likelihood-based phylogenetic inference, a communication bottleneck typically arises when applied to large-scale systems and problems. It is the aim of this work to overcome this bottleneck.

2.5.2 Communication Bottleneck

The majority of parallel maximum likelihood applications available employ the master/workers approach [49]. With this approach, generally one processor serves as the master and coordinates multiple worker, or slave, processors. The likelihood calculation, including branch length optimization, for each tree is the parallelizable entity. Thus trees are evaluated in parallel during each 'round' since the calculations for a given tree do not depend on other trees evaluated in that round. Recall that during each round, in the fastDNAml algorithm for instance, a certain number of topologies are formed, and for each the likelihood is calculated, followed by a comparison of the results. The tree producing the maximum likelihood then serves as the starting tree for the next round of topology generation. The master is responsible for forming the new topologies, which are then sent out one by one to the workers. Once each worker has calculated the likelihood for a tree, the results are sent back to the master, and the worker may receive another tree from the master for evaluation if any remain. After all results for the round have been received, the master determines the tree with the maximum likelihood.

As parallel systems with more processors become available, the master/workers scheme increases the number of workers, which all share the single master. A communication bottleneck arises as the master struggles to handle the number of outgoing and incoming messages. Workers become idle as they wait for the master to send topologies needing evaluation, and thus the full potential to be gained through parallelization is not realized. This bottleneck renders applications using the master/workers technique suitable only for small-scale parallel systems, therefore limiting the size of tree that can be computed in a reasonable amount of time. A solution is needed in order for researchers to take advantage

of the steadily increasing amount of sequence data and accessibility to supercomputers, which can have thousands of processors.

While avoidance of the bottleneck has recently been addressed with the introduction of new algorithms [49], biologists tend to distrust unfamiliar computational methods and instead prefer to use results from older and more well-known programs [31]. Therefore this work uses the existing popular sequential program fastDNAml as a foundation for parallelization, with the aim of achieving comparable results. While this program may not use the most advanced algorithms available, it nonetheless provides an acceptable foundation as the focus of this work is not the quality of the results, but instead relief of the communication bottleneck. The master/workers scheme has been adapted via several optimizations that now make it suitable for large-scale parallel systems and problems. This has been accomplished in several ways. First, multiple trees are packed into a single message to be sent to a worker for evaluation, reducing the number of messages sent by the master, along with the associated overheads. Secondly, the number of messages received by the master was reduced by having the workers only return their best trees at the end of each round. Both modifications assume each worker has many trees to evaluate in a given round. Finally, the master's responsibilities of topology generation, communication with the workers, and results comparison are spread over multiple processors, with each assigned to a subset of the total number of workers. The specifics will be presented in a later chapter. These optimizations provide the research community with a way to improve existing, more advanced programs that are inhibited by a traditional master/workers approach.

The ultimate goal is to combine this work with other programs currently under development in the research group to form a suite of parallel programs utilizing various methods of phylogenetic inference. This will offer more flexibility to the user along with the ability to compare the best trees determined through differing criteria. The package will be free license and open-source, making the source code available for modification by the user or incorporation into an existing program. In addition to maximum likelihood, the package will offer the maximum parsimony and disk-covering methods (DCM). DCM have the potential to use any phylogenetic inference method as the base method, though initially neighbor joining will be the only option available. It is the hope that this work will benefit many applications of phylogenetic inference, including drug discovery, disease tracking, forensics, and protein function determination [3] [6].

2.6 Existing Parallel Efforts using ML

Phylogenetic inference is an ongoing area of research, with researchers aiming to create implementations yielding increasingly more accurate results in a reduced amount of time. Several parallel implementations of the maximum likelihood method have been developed over the past decade in an attempt to achieve these aims, however they suffer from the limitations discussed above. These parallelizations have been mostly based on the master/workers scheme, several of which are presented below. Other approaches are also discussed.

Parallel fastDNAml [38], or pfastDNAml, is a parallel implementation of the sequential fastDNAml method using the Message Passing Interface (MPI), and was considered a state-of-the-art parallel maximum likelihood program in 2001 [37]. The parallel algorithm

is based on the high computation-to-communication ratio for evaluation of individual tree topologies. Evaluating the branch lengths and likelihood of each tree is the most computationally intensive step in the maximum likelihood process [35], whereas there is relatively little to communicate with worker processes, mainly the topology and calculated branch lengths and likelihood values. A master process is responsible for generating the trees to be evaluated, which are then transferred to a foreman process. The foreman dispatches a tree to each worker process, which returns the tree with its associated branch lengths and likelihood values back to the foreman once they are calculated. The worker then receives another tree from the foreman for evaluation. The foreman compares the tree likelihoods at the end of each step, and returns to the master the tree with the maximum likelihood, to which an additional taxon is added, and so on. Topological rearrangements are performed by the master on the best tree resulting from each step. These rearranged topologies are then dispatched to the worker processes by the foreman, with the best tree returned to the master for further rearrangements if the likelihood continues to increase, otherwise another taxon is added. The program flow for parallel fastDNAml is provided in Figure 2.5 below.

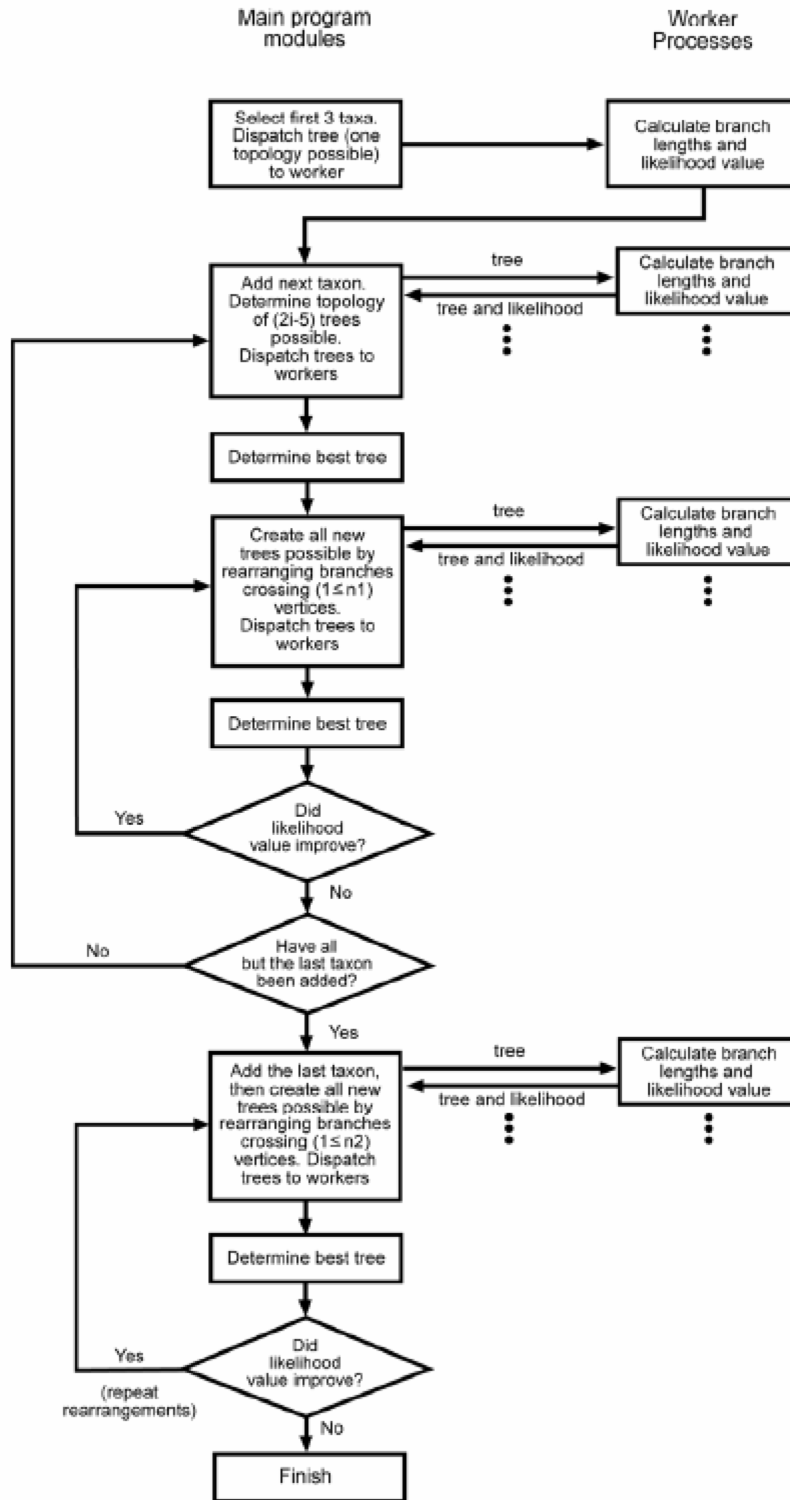


Figure 2.5: Parallelization of fastDNAml (pfastDNAml) [38].

TREE-PUZZLE [30] is a parallelization of the quartet puzzling algorithm where the *ML* and *puzzling* steps are parallelized, as shown in Figure 2.6. The program was implemented using MPI and a master/workers scheme, with load-balancing achieved through a modified version of the Guided Self-Scheduling algorithm.

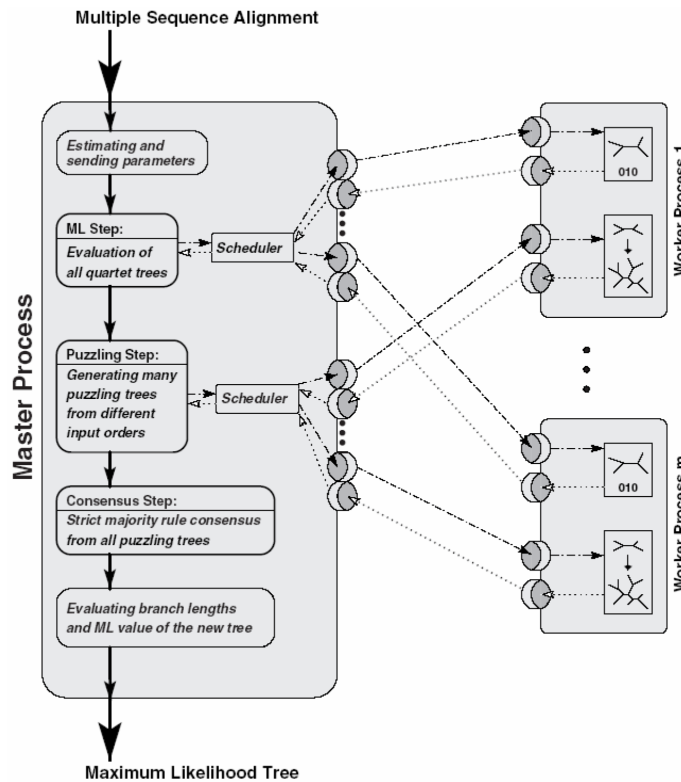


Figure 2.6: TREE-PUZZLE parallelization [30].

The pIQPNNI program [21] is an improved and parallelized MPI version of IQPNNI. The initial local optimum is found sequentially with BIONJ and NNIs as in IQPNNI, with the parallelization applied to the random deletions and reinsertions of leaves into the tree, as well as the associated NNIs. It is these processes that occupy over 90% of the runtime in the improved sequential version of IQPNNI, and thus warrant parallelization. Figure 2.7 shows the parallelization scheme. A modification to the sequential algorithm

had to be made for parallelization since each progressive iteration of deletions, reinsertions and NNIs uses the new best tree found from the previous iteration. Once the initial optimized tree is found sequentially it is sent by the master to all worker processors, which each then carry out their own deletions, reinsertions and NNIs. When a worker has found its optimal maximum likelihood tree, it is sent back to the master who then updates the current best tree if the likelihood has improved. This new best tree is then broadcast to all workers, who start optimizing the new tree once the optimizations of their current tree are completed and returned to the master, thus overlapping communication and computation. The master is responsible for alerting all workers to stop at the appropriate time.

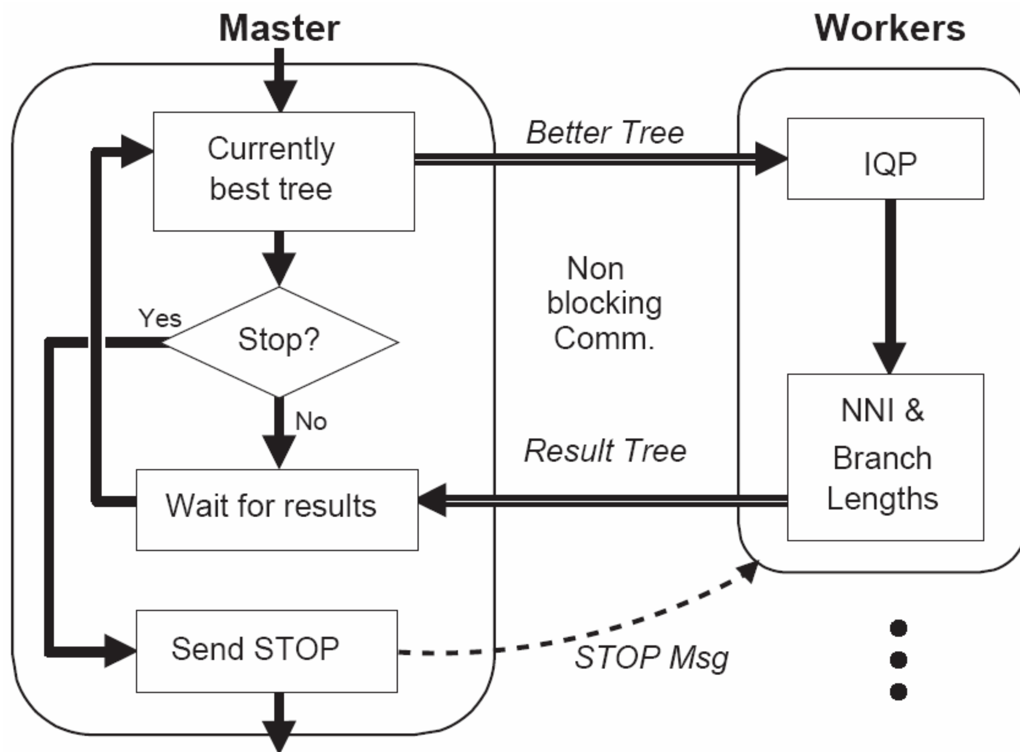


Figure 2.7: pIQPNNI parallelization [21].

A parallel version of the genetic search algorithm has been implemented in GARLI using MPI, yielding improved results over the sequential version for large datasets [51] [50]. This is due to the evolution of independent populations on each processor, which leads to more diverse individuals, or trees. However, this does not necessarily result in significantly faster run times over the sequential algorithm. Each “remote” population communicates with the “master” population, but the remote populations do not communicate among themselves. The master population periodically receives the top tree from each of the remote populations. These trees then serve as parents of the next generation or are used to form new individuals through recombination, involving a subtree interchange between two trees. The master also uses the best scoring tree over all populations to reseed remote populations whose top tree scores less than the best overall tree by an amount greater than a specified threshold.

An OpenMP parallelization of RAxML-V, RAxML-OMP, was developed for Symmetric Multi-Processing machines [37]. Loop-level parallelization was employed at each of the three main operations, specifically likelihood vector computation, branch length optimization, and computation of the final tree likelihood value. Each of these basic functions utilizes a loop in RAxML-V, with one iteration per nucleotide site. Each loop iteration is therefore independent, and different processors are assigned to different sections of sites, and thus independently access different regions of main memory.

Parallelization has also been applied to RAxML-VI-HPC, though only for the purposes of bootstrapping and generating multiple final trees from unique starting trees [34]. MPI was used for the parallelization on PC clusters.

While no official parallel implementations of PHYML have been found in the literature, A. Stamatakis, who is responsible for developing RAxML, claims to be working on an OpenMP version of PHYML [37]. An unofficial parallel version of PHYML has also been implemented [44].

Background information has been presented regarding phylogenetic inference, the associated maximum likelihood method, and the inadequacies of previous related parallelization efforts. The significance of this thesis work is now evident. From this point, the discussion begins to narrow to the specifics of the approach taken to accomplish the goals of this thesis. As the parallelization performed for this work was built from an existing sequential foundation, a description of that foundation will be provided in the following chapter, before discussing the details of the parallelization.

Chapter 3 fastDNAml

The previous chapter provided general background information pertaining to phylogenetic inference and the maximum likelihood method, along with a discussion of how this thesis contributes to that area of research. Additional background information, more closely related to the specifics of this thesis, is provided in this chapter. Here, the fastDNAml program is introduced, which serves as the sequential foundation for the parallelization presented in the next chapter. The reasoning behind selecting fastDNAml as a foundation is given, along with details of its implementation. Several modifications made to the program in preparation for parallelization are also described.

fastDNAml is a sequential computer program for phylogenetic inference utilizing the maximum likelihood method. Olsen, et al. [23] developed fastDNAml in 1994, building on version 3.3 of the program DNAML written by J. Felsenstein [8]. DNAML is currently offered as part of the popular PHYLIP package of phylogenetic inference programs [10]. Several improvements over DNAML were incorporated into fastDNAml to achieve faster execution, focusing mostly on branch length optimization. One such change was a switch from the EM method of branch length optimization to the Newton-Raphson method [23]. The most current version of fastDNAml, version 1.2.2, was released in 2000 and provides the foundation for this work. Updated versions of DNAML have come out since the introduction of fastDNAml with changes along similar lines.

3.1 Foundation for Parallelization

The parallelization and subsequent optimizations presented in this work were built from a foundation provided by sequential fastDNAml. Numerous aspects made fastDNAml ideal for this role. Most importantly it satisfied the requirement of using the maximum likelihood method. Also, the source code is freely available for download and written in C, facilitating incorporation of the Message Passing Interface (MPI) routines for parallelization. With its extended presence in the field it has gained popularity among researches who place trust in the results. By achieving these same trusted results it is the hope that researchers will not hesitate to embrace the optimizations introduced. Further, because fastDNAml has served as a foundation for other more advanced phylogenetic inference programs, these optimizations may be more easily incorporated into those works. Compared with other maximum likelihood-based phylogenetic inference programs, fastDNAml is considered less complex, making it more amenable to modification. While an official parallelization of fastDNAml exists, pfastDNAml [38], fastDNAml was parallelized from scratch for this work and followed by optimization. Although both make use of the master/workers scheme, somewhat different approaches were taken to parallelization. The two will be compared in Chapter 4.

Originally, the intention of this thesis was to produce a parallel maximum likelihood-based phylogenetic inference program written from scratch without employing an existing sequential program as a foundation. However, it became apparent that owing to the complexity of the maximum likelihood method a considerable amount of time would be required to complete such a task. This could not be validated for a Master's level

thesis, especially when considering that the focus was on parallelization and related optimizations rather than details of the maximum likelihood method itself.

3.2 Implementation

What follows is a description of the implementation of Olsen's sequential fastDNAml program. These details will aid in understanding the parallelization and optimizations of fastDNAml presented in later chapters.

3.2.1 I/O Format & User Options

Input and output occur through standard input (stdin) and standard output (stdout), respectively, with the formats based on those for DNAML. Input begins with the number of sequences and the number of sites. Any user options are then entered, followed by the species' sequence data. Principle user options are provided in Table 3.1 below.

User option	Purpose
Categories (C)	Assigns site-specific rates of evolution. The user creates multiple rate 'categories' and assigns each site (alignment column) to a category. Absence of the 'C' option assigns a default rate of 1 to each site.
Empirical Frequencies (F)	Selects between use of user-derived base frequencies and empirical base frequencies (derived from the sequence data). Empirical base frequencies are the default.
Global (G)	Specifies the number of branches to cross during local (partial tree) and global (full tree) rearrangements. The default is 1 branch for both. Note that on rearranging, topologies are created by crossing up to and including the specified number of branches.
Interleaved (I)	Specifies if the interleaved format is used for sequence input. Not interleaved would require all sequence data lines for a species to appear before the next species' sequence.
Jumble (J)	Used to enter random number seed to randomize addition order of input sequences during sequential addition phase. Order of sequence addition to growing tree can affect topology results since heuristic method used. Default order is that in input file. Many runs of program w/ same input file but different addition orders can be accomplished with the 'J' option.
Keep multiple best trees (K)	User can opt to keep more than just the 1 best tree (default) in the list of best trees. The top 'K' tress ('K' highest likelihood scores) are kept.
Outgroup (O)	Selects the sequence number, based on their input file order, used for the tree root when drawing the output tree. The default is to use the first sequence in the input file. This option only affects the drawing of a tree since trees are unrooted internal to the program.
Transition/Transversion ratio (T)	Indicates user-entered transition/transversion ratio. A default of 2.0 is otherwise used.
Weights (W)	Assigns each site (alignment column) a specific weight. Otherwise a default of 1 is used. A site's weight is the factor by which it affects the likelihood result. A weight of 0 implies that the site does not affect the results.
Write tree (Y)	Selects the option to write the best tree to a (.PID) file (distinct from the standard output) using 1 of 3 tree representation formats – Newick, Prolog, and PHYLIP. By default the tree is written and the PHYLIP format used.

Table 3.1: fastDNAml user options.

fastDNAml writes to standard output as the program is running, initially echoing some of the information input. A progress log follows, indicating when each species is added to the

tree, the number of rearranged topologies tested on each round, and the resulting natural logs of the likelihoods. The total number of topologies evaluated is then reported, along with the top tree topology, its branch lengths, and associated likelihood result. If the 'K' user option was entered, the topology, branch lengths, and likelihood results for all top trees is reported. In addition, the results from a Kishino-Hasegawa test [20] are given which identify the top trees that are significantly less probable as compared to the top tree.

More detailed information regarding the I/O formats used for the software associated with this thesis, which is based on fastDNAml I/O, will be provided in a later chapter along with example I/O files. The reader is also referred to the fastDNAml and PHYLIP user documentation [24] [10].

3.2.2 Data Structures

The data used by fastDNAml are organized into structures (structs), which are operated on and passed around by the functions of the program. The major structures are described in Table 3.2 below.

Structure	Description
likelivector	Contains the conditional likelihood values, one for each of the 4 bases, for a particular site at the 'node' struct that owns the 'xarray' struct containing this 'likelivector.'
xarray	Contains an array of 'likelivector' structs, one for each site at the 'node' struct owning this 'xarray.'
node	Represents a node of a tree and contains references to 2 neighbor nodes via 'next' and 'back'. This 'node' struct has a value for the branch length between itself and neighboring node 'back,' as well as an 'xarray' struct for this tree node. If representing a tip node, this 'node' struct holds the species name and a pointer to the associated sequence data stored in the 'rawdata' struct.
tree	Represents a phylogenetic tree. Contains references to the 'rawdata' and 'cruncheddata' structs as well an array of all 'node' structs representing the nodes of this tree. Also holds values for the total number of species (tips), the number added to this tree so far, degree of branch swapping explored so far, a flag indicating all branch lengths are optimized (tree smoothed), the number of branches to cross during local and global rearrangements, and the value of its ln(likelihood).
bestlist	Represents the list of the top scoring trees (based on likelihood), with each top tree saved as a 'topol' struct. It has a value for the maximum number of trees to keep in the list and a flag indicating if the likelihood has improved (i.e. a better tree was found) over the last round of evaluations.
topol	Abbreviation for topology. 'topol' structs are the storage form for 'tree' structs in the 'bestlist' struct. Thus 'topol' is similar to a 'tree' struct and contains the number of species added to the tree so far, the degree of branch swapping explored, a flag indicating all branch lengths are optimized, and the value of its ln(likelihood). A 'topol' also knows its position in the sorted list of top trees ('topol's) in 'bestlist.'
rawdata	Contains input data for the program such as the number of species and sites, base frequency values, transition/transversion ratio and other values related to the model of base substitution, site-specific weight and rate category values, and a 2D array holding the sequence data.
cruncheddata	Contains values related to the crunching (and uncrunching) of the sequence data, including site-specific weight and rate category values for the crunched data.
analdef	Contains user option data such as flags for the use of empirical vs. user base frequencies and input sequence format, the value for the jumble random number seed, etc.

Table 3.2: Data structures of fastDNAml.

The fastDNAml program uses one of each of the 'analdef,' 'rawdata,' and 'cruncheddata' structs to hold information associated with the program as a whole. There is a single 'tree'

struct that represents the tree currently being built and evaluated. The top ‘K’ trees found so far are saved in a single ‘bestlist’ struct, where each tree is represented as a ‘topol’ struct.

The configuration of a tree is represented in the following way. Each tip node, or leaf, of the tree is represented by a single ‘node’ struct containing a reference to the sequence data for the associated species. The reference to the neighbor ‘back’ in the ‘node’ struct for a tip is for the internal tree node to which the tip node is attached. The ‘next’ reference is equal to NULL for a tip ‘node’ struct. Bifurcating trees are used in fastDNAml, where each internal node is attached to three neighboring tree nodes, specifically an ancestor and two descendents. To represent this in a manageable fashion, each internal tree node is composed of three connected ‘node’ structs. These are connected in the form of a triangle where each ‘node’ struct has a reference to one neighboring ‘node’ struct via ‘next.’ Each of the three ‘node’ structs representing a single internal tree node also has a ‘back’ reference to either a tip ‘node’ struct or one of the three ‘node’ structs of it’s neighboring internal tree node. These connections are demonstrated in Figure 3.1 below for a portion of a tree. Here, the ‘node’ structs p1, p2, and p3 together represent a single internal tree node. The ‘node’ struct q represents a tree tip node.

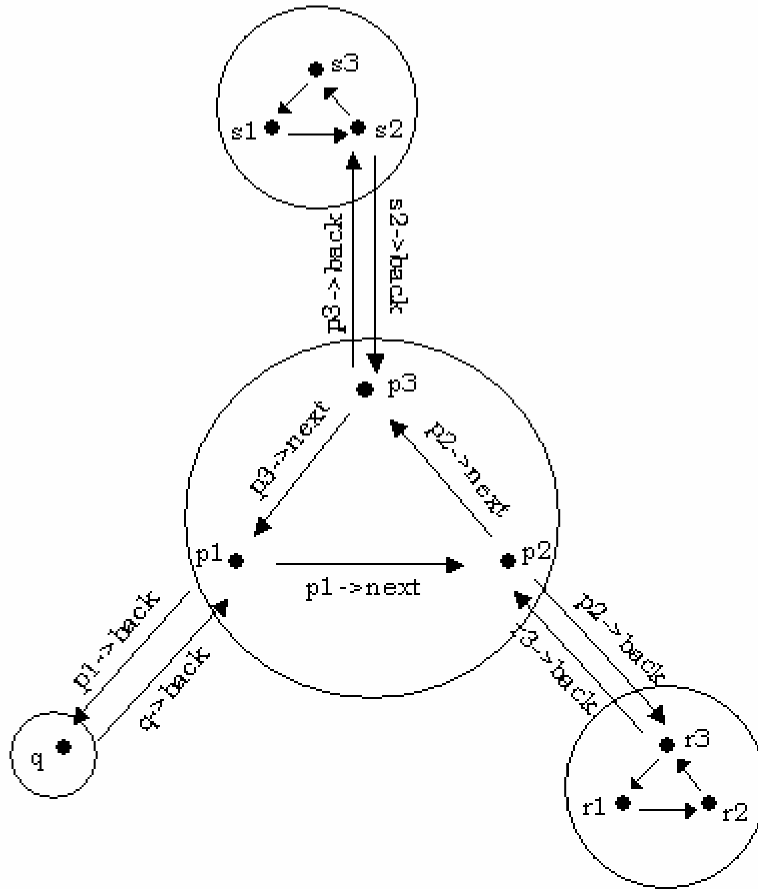


Figure 3.1: Partial tree showing representation of tree nodes with 'node' structs. Circles represent tree nodes, and dots represent 'node' structs.

Note that the branches of a tree are not represented by a dedicated struct, but instead are implicit in the 'back' references between 'node' structs. Branch length values are contained in the 'node' structs. All of the 'node' structs comprising a tree are stored in an array in the associated 'tree' struct.

The 'cruncheddata' struct holds values associated with the crunching of the input sequence data, which is performed before tree building starts. This involves a consolidation of identical sites (columns in a sequences alignment), i.e. sites with the same base and rate category, into a single representative site (column) referred to as a pattern. The tree

building process and likelihood calculations then operate on this crunched sequence data. Without crunching, identical conditional likelihood calculations would be repeated for each of the identical sites, thus wasting computational resources. Recall that the conditional likelihoods for each site are calculated individually. The site-specific weight values are summed over the crunched sites so that their contribution to the final tree likelihood value is unaffected by the crunching. Thus crunching achieves a reduction in computation time without altering the results.

3.2.3 Program Flow

The algorithm used by Olsen's sequential fastDNAml program has been described earlier (see Section 2.4) and is diagramed in Figure 3.2 below. It entails sequential addition of taxa followed by rearrangement, with the maximum likelihood tree used as the starting point for the next taxon addition. This is a heuristic method intended to reduce the search space, and thus not guaranteed to find the best tree. It is thought, however, that if the best tree is not found after each taxon addition that there is a good chance of finding it through rearrangements [42]. Rearrangements are a hill-climbing technique guaranteed to only increase the likelihood, as they are stopped as soon as the likelihood ceases to improve [23].

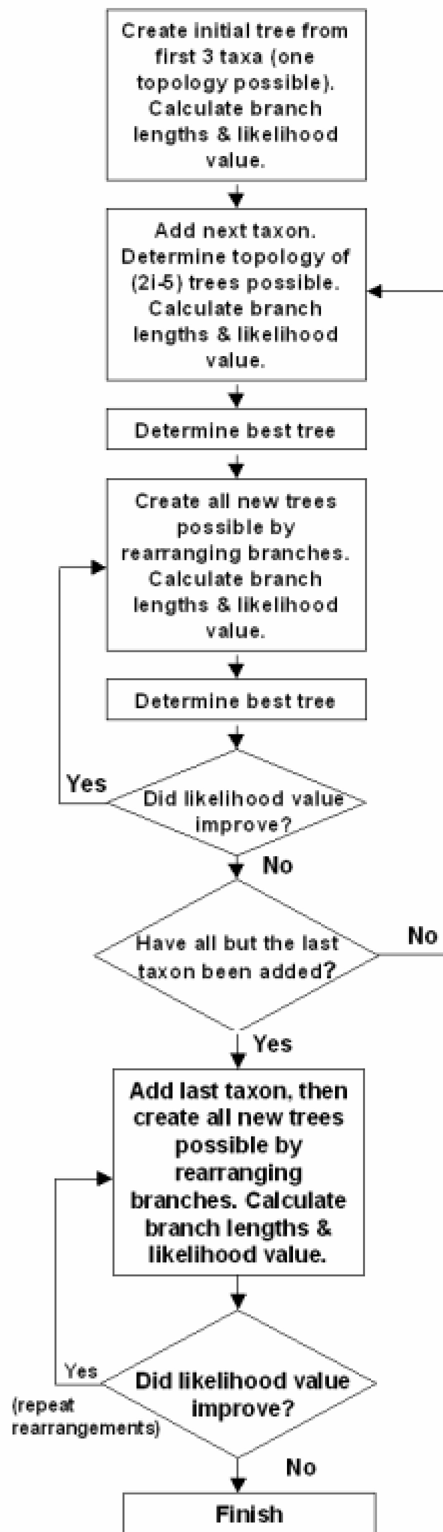


Figure 3.2: Sequential program flow for fastDNAml where 'i' indicates the number of taxa in the current tree. Modified from [38].

This flowchart translates into the pseudo-code provided below in Figure 3.3. Program execution begins with main(), which mostly performs initializations along with a call to the function makeDenovoTree(). It is this function, whose pseudo-code is given below main(), that is the heart of the fastDNAml algorithm and depicted in the flowchart of Figure 3.2. Pseudo-code for functions called by makeDenovoTree() is also provided in Figure 3.3.

```
main() {
 //initializations
 getinput()         //read input, initialize
 makeweights()      //crunching
 makevalues()       //cont crunching
 empiricalfreqs()   //calc empirical base freqs
 reportfreqs()      //calc values for model of base sub
 linkdata2tree()    //link seq data to tree (tree currently has 0 species)
 linkxarray()       //setup 'xarray' structs
 setupnodex()       //cont setup 'xarray' structs
 initBestTree()     //init list of best trees ('bestlist' struct)

 //build & evaluate trees
 makeDenovoTree()
} //end main


makeDenovoTree() {
 buildSimpleTree()   //build initial tree of 3 taxa
 while(more species to add) {
    buildNewTip()     //prep next taxon to add
    resetBestTree()   //clear list of best trees
    addTraverse()     //eval all (2i-5) insertion points (trees)
    recallBestTree()  //get best resulting tree
    optimize()          //rearrangements (partial or global)
 } //end while
 showBestTrees()     //output results
 cmpBestTrees()      //compare multiple best trees (if > 1 best tree)
} //end makeDenovoTree
```

```
addTraverse(node p, node start) {
 //node ‘p’ to be inserted into tree
 //node ‘start’ indicates insertion location (start of recursion)
 //select ‘start’ carefully for traversal of entire tree
 //traversal only moves forward (i.e. not in direction of start->back)
 testInsert(p, start)  //insert p and evaluate, then undo
 if(p != tip) { //continue traversing tree in forward direction
  addTraverse(p, start->next->back)          //repeat for neighboring branch
  addTraverse(p, start->next->next->back)    //repeat for neighboring branch
 } //end if
} //end addTraverse


testInsert(node p, node q) {
 insert(p, q)  //inserts p at branch q:q->back, optimizes br lengths, calcs cond likelihoods
 evaluate()   //calcs overall likelihood of tree
 saveBestTree()  //saves new tree to list of best trees if worthy
 hookup()     //remove p to restore orig tree
 restoreZ()   //restore orig br lengths
 initrav()    //regenerate orig cond likelihoods
} //end testInsert


optimize() {
 do {
  startOpt()          //remember starting tree
  rearrange()         //one round of branch swapping
  recallBestTree()  //get best resulting tree to use for next round
 } while(best tree is not starting tree)  //continue while likelihood improving
} //end optimize


rearrange(node p) {
 //p is node in tree where rearrangements begin
 q = p->back

 //moving subtree forward in tree
 if(p != tip) {
  p1 = p->next->back
  p2 = p->next->next->back
  //remove p to form tree containing p1,p2,etc and subtree containing p,q,etc.
  removeNode(p)
  if(p1 != tip) {
   //eval insertion of subtree in all locs forward of p1->next->back
   addTraverse(p, p1->next->back)
   //eval insertion of subtree in all locs forward of p1->next->next->back
```

```
    addTraverse(p, p1->next->next->back)
  }
  if(p2 != tip) {
    addTraverse(p, p2->next->back)
    addTraverse(p, p2->next->next->back)
  }
  //restore orig tree
  hookup()
  initrav()
} //end if(p != tip)

//moving subtree backward in tree
if(q != tip) {
  q1 = q->next->back
  q2 = q->next->next->back
  //remove q to form tree containing q1,q2,etc. and subtree containing q,p, etc.
  removeNode(q)
  if(q1 != tip) {
    addTraverse(q, q1->next->back)
    addTraverse(q, q1->next->next->back)
  }
  if(q2 != tip) {
    addTraverse(q, q2->next->back)
    addTraverse(q, q2->next->next->back)
  }
  //restore orig tree
  hookup()
  initrav()
} //end if(q != tip)

//move other subtrees
if(p != tip) {
  rearrange(p->next->back)
  rearrange(p->next->next->back)
}

} //end rearrange
```

Figure 3.3: Pseudo-code for sequential fastDNAml. Pseudo-code for main() is located at the top, followed by the pseudo-code for makeDenovoTree(). Pseudo-code for several functions called by makeDenovoTree() is also provided.

3.3 Modifications

Several modifications were made to sequential fastDNAml in preparation for incorporating the parallelization and optimizations associated with this thesis work. The main goal was to simplify the program in order to facilitate both the implementation itself and the explanation of the implementation to be provided in later chapters. This involved the removal of portions of code that were not essential to demonstrating bottleneck relief upon parallelization. The core functionality of fastDNAml, as described in the previous section, was not changed.

To start, several user options were discarded. These included option 'U,' which allowed input of user-supplied trees for evaluation, along with option 'L' for input of user branch lengths for the user-supplied trees. The 'R,' or restart, option was also eliminated, which allowed the user to input a tree for modification such as insertion of additional taxa or further rearrangements. Another option deleted was the quickadd feature, option 'Q', which would cause only the few branches local to an insertion point to be optimized on insertion of a new taxon, rather than all branches of the tree. This would have proved too difficult to incorporate into the parallelization. Option 'F' indicating user-entered base frequencies was not removed, but modifications were made to fix a bug discovered that prevented reading in other user option lines under some circumstances.

Other changes include the addition of code for the reading of input from a file, which was made the default rather than standard input. Automatic writing of 'checkpoint' trees to a checkpoint file after each taxon addition and round of rearrangements was removed. Code associated with parallelization using p4 tools and loop-level vectorization, incorporated

into fastDNAml as options, was also removed. Many comments were added since the code comprising fastDNAml, filling over 70 pages, was scarcely commented. This was a very time-consuming task as the code was difficult to decipher, which made the addition of comments all the more necessary. Other minor code changes were made that are not worth elaborating on, such as the addition of missing statements for memory deallocation, etc.

The fastDNAml program described in this chapter provides an ideal foundation for parallelization as it uses the typical algorithm for maximum likelihood-based phylogenetic inference and its results are trusted among biologists. The implementation details that have been presented will facilitate understanding of the parallelization presented in the next chapter, as well as of the subsequent optimizations. This marks the end of the background material, with the focus of this document now shifting to the work performed as part of this thesis.

Chapter 4 Parallelization

Once the modifications of fastDNAml were completed, a parallelization of the code was carried out as part of this work. Although not remarkable in itself, it laid the framework on which to incorporate the optimizations that form the core of this thesis. Parallelization allows the computational problem to be spread over multiple processors, thus decreasing the execution time which otherwise becomes impractical for large phylogenetic inference problems utilizing maximum likelihood. As discussed previously, when the typical master/workers scheme is employed a communication bottleneck arises, which will be demonstrated by this parallelization. The subsequent optimizations described in a later chapter aim to relieve this bottleneck. Arriving at the same trusted results as fastDNAml in a shorter time is the ultimate goal.

4.1 Approach

Parallelization was accomplished using the master/workers scheme, where one processor serves as the master and multiple other processors are each workers. The master orchestrates the distribution of computation over the workers, which execute in parallel. This leads to a reduction in execution time for a given problem relative to its sequential execution, which generally improves as the number of processors, and thus workers, increase. The most often parallelized entity with the maximum likelihood method of phylogenetic inference is the branch length and likelihood calculation of an individual tree, and is what is utilized here. The calculation of branch lengths and likelihood for an individual tree is independent of other trees evaluated in a given round. Thus the calculation for one tree can be assigned to a single slave, and communication is not needed

among slaves. Once initialized, a slave needs only a topology, for which it can then produce the associated branch length and likelihood results.

Implementation of the parallelization was accomplished with the message-passing programming model, where processors communicate via the sending and receiving of messages. In this application text strings representing trees, including the associated branch lengths and likelihood, are passed between the master and workers. For each topology to be evaluated, the master sends the target worker a message containing that topology and its preliminary branch lengths and likelihood. Upon completing its calculations, the worker returns the topology to the master with its final branch lengths and likelihood. Since the branch length and likelihood calculation is the most computationally intensive part of maximum likelihood-based phylogenetic inference [35], with relatively little to communicate between processors, a high computation-to-communication ratio exists for an individual tree's evaluation. Higher ratios lead to better performance in parallel processing.

The algorithmic flow of fastDNAml was maintained for the parallelization carried out in this work. The official parallelization of fastDNAml, pfastDNAml [38], also follows this same algorithmic flow, and the reader is referred to the associated flowchart in Figure 2.5. Note, however, that while the algorithm is shared between the two parallelizations, the implementations differ. For instance, the 'main program modules' of Figure 2.5 represents both a master process and a foreman process in pfastDNAml, whereas a single master fulfills both roles in this work. Differences between the two implementations will be discussed further in a later section. From Figure 2.5 it can be seen that there are three instances where a set of trees undergoes evaluation by the workers. This excludes the single

3-taxon tree, which in this work is evaluated by the master. The first of the three instances follows insertion of the ith taxa, where (2i-5) topologies are evaluated corresponding to the (2i-5) possible insertion points. Each of these topologies is sent out to a worker by the master, which selects the top-scoring tree from the results. Another set of topologies requiring evaluation is then created by performing local rearrangements on this top-scoring tree. Several rounds of rearrangements are performed, each time starting from the top tree of the previous round as determined by the master, until rearrangements no longer improve the likelihood of the top tree. The same number of topologies is evaluated on each round, and is dependent on the user-selected number of branches to cross on local rearrangements as well as the current number of taxa in the tree. The third instance of topology evaluation by the workers also involves rearrangements, but on a much larger, global scale. This occurs after the final taxon has been inserted into the tree and the resulting top tree has been determined. Rather than performing the usual local rearrangements after insertion, global rearrangements are carried out instead. Typically the user-selected number of branches to cross on global rearrangements is much larger than that for local rearrangements, resulting in a much greater number of possible topologies to evaluate on each round. Once the global rearrangements cease to improve the likelihood, the maximum likelihood tree is taken as the program result, possibly also along with other high-scoring full trees as desired by the user.

The procedure followed by the master and workers to evaluate a set of topologies in parallel is the same whether following a new taxon insertion or rearrangements. Each worker has only a single instance of a tree at any time, which is updated for each new topology to be evaluated. The master only works with one tree at a time as well, which is

either the best tree found so far or a new topology being built from the starting tree. The master also stores the best-scoring K trees evaluated so far in the round in its ‘bestlist’, where K can range from 1 to any user-selected number of trees. fastDNAml was designed such that following generation of a new topology, the branch lengths and likelihood are evaluated, and based on the resulting likelihood the tree is saved to the ‘bestlist’ if worthy, or discarded if its likelihood is less than the top K found so far. The original starting tree is then restored in preparation for generation of another new topology. With parallelization, however, the master does not receive any evaluated trees until all topologies for that round have been generated and sent out to the workers. This can be seen in Figure 4.1 below which shows the sequence of sending and receiving trees between the master and three workers for a single complete round of evaluations involving five trees.

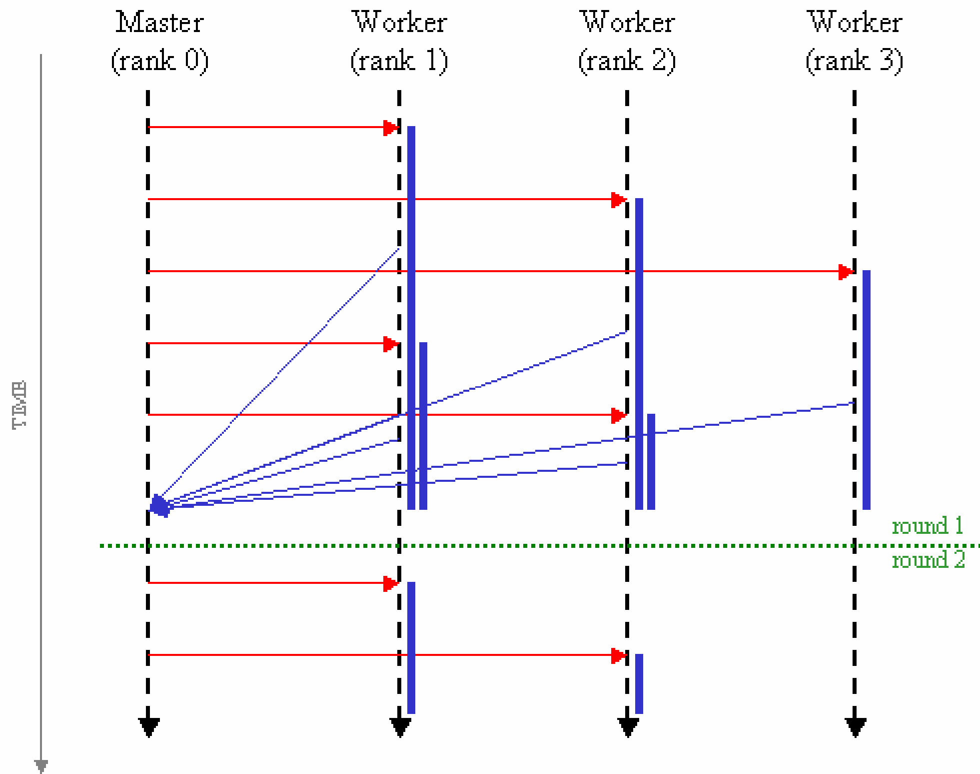


Figure 4.1: Sequence of sending and receiving trees between processors for one round of evaluations. The red arrows indicate a message containing a tree needing evaluation, and blue arrows indicate a message containing the results of a tree's evaluation. The vertical blue bands represent the time range during which a worker may send results to the master. A worker however must compute and send the results of a previous tree before evaluating the next tree.

The master keeps track of only the number of topologies it has sent out, rather than of the specific topologies themselves, and waits to receive this same number of evaluated trees before continuing on. The trees are received in the order that they are returned from the workers, which may or may not be the order that they were sent out by the master. The order received, and thus compared, generally does not have an effect on the results, although there are some circumstances where this may not be the case as discussed later in this chapter. Upon receipt of each evaluated tree, the master compares the likelihood with those of the top K trees in its 'bestlist.' If the likelihood is among the top K received so far, or the master has not yet received K trees, the new topology is added to the 'bestlist', and

otherwise discarded. The next topology is then received and similarly compared until all trees have been received by the master. Since the maximum likelihood tree of the previous round serves as the starting tree for subsequent rounds of topology generation, the master must wait to compare all evaluated trees before the best tree of the round can be asserted. All slaves are therefore assured to have completed their calculations before the program continues, thus creating a synchronization barrier.

Immediately following generation of each new topology in a given round the topology is sent out by the master to the target worker. The target worker is selected based on the ordering of the workers by their rank, which is a number uniquely identifying each worker. A worker's rank is a value from one to one less than the total number of processors. A rank of 0 is reserved for the master processor. On each round the first topology is sent to the worker with rank one, with the rank of the target worker increasing by one with each new topology sent. The master is aware of the total number of workers, and once all workers have been sent a topology the master continues sending topologies again with rank one. This is demonstrated in Figure 4.1. It can be the case, especially early on when topologies contain few taxa, that there are more workers than topologies to evaluate. In this instance higher-rank workers go unused. In a given round it is assumed that a comparable amount of time is needed for the evaluation of each topology. Therefore no extra effort is made to balance the workload evenly over the worker processors. Achieving a smaller grain size by dividing up the branch length and likelihood calculations of an individual tree would not be likely to result in an improved performance. This is due to the inter-dependencies of the calculations, which would lead to significant communication overheads.

The only role of the worker processes is to calculate the branch lengths and likelihoods of topologies received from the master. Workers are unaware of what stage the program is at. Each worker executes a continuous loop of waiting, receiving, evaluating, and sending topologies. Topologies are sent out to the master immediately after their evaluation is complete, even though the master may still be generating new topologies and thus not yet ready to receive the results. This is not an issue as messages are held in buffers external to the program until the receiving process makes an explicit call to receive a message. These buffers are also utilized in the situation where a worker has not finished evaluation of a previous topology before it is sent another topology awaiting evaluation. The vertical blue bands associated with each worker in Figure 4.1 represent the time range during which results corresponding to the topology received may be sent back to the master. As illustrated, a message containing another topology may arrive before a worker has completed evaluation of a previous topology, in which case it waits in a buffer until explicitly received.

4.2 Computational Resources

The code enhancements made for this thesis work were written in C using MPI, and the program was executed on the RIT Research Computing Cluster. The C programming language was selected for implementation of the parallelization and subsequent optimizations for two main reasons. First, the original fastDNAml code providing the foundation for this work was previously written in C. Secondly, MPI provides message-passing routines designed for incorportion into C code. MPI, or the Message Passing Interface, is a popular library specification used to achieve parallelization in message-

passing environments such as clusters. There are a handful of function calls that can be made from within a C program. The two most often used routines in this work are MPI_Send() and MPI_Recv(), which send and receive, respectively, specified data to or from a specified processor. These two functions offer an asynchronous, blocking mode of communication. A call to MPI_Send() blocks until the message has been sent, but does not wait for the message to be received by the destination process. A call to MPI_Recv() blocks until the message is received. For each MPI_Send() in a program, there is a corresponding MPI_Recv() executed by the destination process. MPI offers several other useful routines such as MPI_Wtime(), which in this work is used for timing program execution. Several versions of MPI exist, with each version having several implementations available. MPICH version 1.2.7 [22] is installed on the RIT Research Computing Cluster and used in this work, which is an implementation of MPI version 1.2. This version of MPI uses static process creation where the number of processes does not change during execution, and is typically selected by the user at program startup.

The code presented in this work was developed and run on the IBM High Performance Computing cluster supported by the Research Computing Department at the Rochester Institute of Technology (RIT) [26]. The cluster consists of a head node and 47 compute nodes, which communicate over a private high-speed network. The head node is composed of dual 2.0 GHz Pentium IV Xeon processors along with 1 GB of RAM. Six 36 GB hard drives are also part of the head node. Each of the 47 compute nodes is composed of dual 1.4 GHz Pentium IV Xeon processors and 512 MB of RAM, along with one 36 GB hard drive. Although there are 96 total processors available, running jobs on the head node is

discouraged, so a maximum of 94 processors was used in this work, equating to 93 workers and one master.

4.3 Implementation

4.3.1 Message Format

Parallelization was achieved using the message-passing model, where communication between processors involves the exchange of messages. The messages specific to this work are mainly text strings representing trees, and are passed between the master and workers. A uniform format is used for both evaluated trees and trees awaiting evaluation. Newly generated topologies to be sent to workers for evaluation are given initial branch length values by the master, which are rough estimates mostly based on values from the starting tree. The master initializes the likelihood to a low 'unlikely' value. In addition to the branch length and likelihood values, information regarding the topology of the tree is also contained in the message string. This is necessary not only for the workers but for the master as well since no record is kept of the topologies sent out and their order as returned from the workers is undefined. The placement of parentheses and commas is used to relay the topology, following the Newick format of tree representation. Refer to Figure 4.2 for a sample message string and its associated topology containing 5 taxa.

```
[&&fastDNAml: version = '1.2.2', likelihood = -
365.8085035374868425606109668, ntaxa = 5, opt_level = 1, smoothed = 1]
(('Sequence5': 0.018731016033274,'Sequence4': 0.067423090414073):
0.087747758741089,('Sequence2': 0.053462492291958,'Sequence3':
0.061771983694398): 0.069398201874276,'Sequence1':
0.082888651788679):0.0;
```

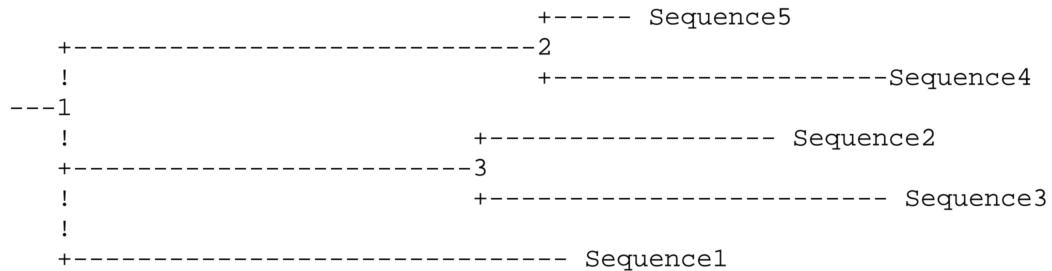


Figure 4.2: Message string and corresponding topology for a tree, as produced by the program presented.

Keep in mind that the tree here has a relatively small number of taxa for demonstration purposes, and that trees typically handled during program execution are much larger, with significantly longer message strings. Each set of parentheses in the string encases the descendent subtrees of an internal node. Within parentheses, commas separate subtrees descending from the same internal node. In a simpler form the topology in Figure 4.2 could be represented by the following string:

(('Sequence5','Sequence4'),('Sequence2','Sequence3'),'Sequence1')

Notice that the two subtrees, 'Sequence5' and 'Sequence4', descending from internal node ‘2’ are contained within a set of parentheses and separated by a comma. Here the term ‘subtree’ is misleading, as 'Sequence5' and 'Sequence4' are species at the tree tips rather than internal nodes with continuing branches. Along similar lines, subtrees ('Sequence5','Sequence4'), ('Sequence2','Sequence3'), and 'Sequence1' all descend from

internal node ‘1’ and are thus contained within the same set of parentheses, separated by commas. The length of each branch in the tree is also included in its message string, and appears after the ‘:’ following the subtree that descends along it. For instance, 0.069398201874276 is the length of the branch between node ‘1’ and node ‘3.’

A header encased in square brackets preceding the tree representation is also part of the message string. This header includes the likelihood of the tree, as well as the number of taxa (ntaxa) in the tree, the degree of branch swapping explored (opt_level), and a flag indicating optimization of branch lengths (smoothed). The program name and version are also included, and for this work the name and version of fastDNAml are kept.

The message string format described here was adopted from the p4 parallelization in the original fastDNAml program, with this same format also later used in pfastDNAml. As such, routines to generate a tree’s corresponding message string (treeString()) and convert it back to a tree upon receipt (str_treeReadLen()) already existed and were incorporated into this work. The routines from the fastDNAml code were found to contain several bugs, and therefore updated versions from pfastDNAml were used instead.

4.3.2 Pseudo-code

The pseudo-code presented in Figures 4.3 and 4.4 shows how parallelization is accomplished through code executed by the master and multiple workers. This parallel code was built on top of the sequential code modified from fastDNAml given in Figure 3.3. As is common with parallelization, the same program code is run on all processors. Different sections of code are designed for execution only by certain processors, such as by the master rather than a worker. Shared portions of code to be executed by all processors

can also be found. The rank of the executing processor, which is known by that processor, is used during runtime to achieve this separation. For instance, rank 0 is considered to be the master processor by convention, and thus would execute a if (rank==0) { } code block, for example. This division by rank is not demonstrated in the pseudo-code provided, but Figure 4.3 displays the code to be executed by the master only, separate from the code in Figure 4.4 to be executed by each worker. The number of workers, which execute the code in Figure 4.4 in parallel, is equal to one less than the total number of processors. The number of processors desired by user is often entered on the command line when the program is invoked. If 10 total processors are selected, for instance, their ranks will range from 0 to 9, where the 9 workers would be assigned ranks 1 through 9.

On program startup, a section of memory allocations and initializations is shared by all processors, both master and workers. Here each processor creates its own dedicated structures as described in section 3.2.2, including ‘tree’, ‘analdef’, ‘rawdata’, ‘cruncheddata’, etc. Originally workers were not given their own ‘bestlist’ structure, but this was changed once the optimizations described in the next chapter were added. In order to properly initialize these structures the master must first read the input file, converting it to a string, which is then sent to each of the workers. Each worker then writes this string to a file in its local memory. This allows all processors to share the code invoking the initialization routines that read input from a file. These initializations are done in parallel. Much of the data in these structures does not change during execution, and assigning each worker its own dedicated structures upfront cuts back on subsequent communication costs. This design was adopted from pfastDNAml, and some of the associated code in this work was modified from pfastDNAml. The function initInputFile() called by all processors was

created to perform the input file transfer and setup. Once set up, the file is used in the initialization routines shown in main() in Figure 3.3.

Following initialization, main() divides into two separate sections for the master process and the workers. The master calls makeDenovoTree() while each worker calls waitForWork(). Both routines contain loops that continue until the final best tree(s) is determined. Once this occurs, the master exits makeDenovoTree() and invokes killSlaves() signaling each worker process to terminate before itself terminating. The routine makeDenovoTree() was modified only slightly from the sequential version in Figure 3.3. One change was the addition of recvTrees(), whose pseudo-code can be found below that of makeDenovoTree(). Once all topologies have been generated and sent out, recvTrees() is called to receive the results. All evaluated trees are received one at a time and saved to the master's 'bestlist' if they are among the top scoring trees. To send trees out for evaluation by the workers the code in testInsert(), called by addTraverse(), was changed, with makeDenovoTree() unaffected. As seen in Figure 4.3, sendTreeforEval() replaces the master's calls to evaluate() and saveBestTree(). Figure 4.3 also includes pseudo-code for sendTreeforEval(), where the newly generated tree is sent to the next worker for evaluation. Notice that 'nextSlave' is reset in makeDenovoTree() so that trees are sent to the worker of lowest rank at the start of each new round.

The insert() routine called from testInsert() was also altered, although its pseudocode is not shown. In sequential execution branch lengths are optimized in insert(), but with parallelization the workers instead take on this responsibility. The master therefore only assigns rough estimates to the branch lengths in insert() before sending the tree off to a

worker. The call to initrav() in testInsert(), which regenerates the conditional likelihoods of the original starting tree for the round, is also no longer necessary as the conditional likelihoods are not changed by the master. The optimize() routine was changed in a similar fashion as makeDenovoTree(), where recvTrees() is called to receive the resulting tree evaluations. The rearrange() and addTraverse() functions were not changed for parallelization.

The call to buildSimpleTree() in makeDenovoTree() creates the initial 3-taxon tree, whose branch lengths are optimized by the master rather than by a worker. This was done since only one possible topology configuration exists for three taxa, and thus only one tree needs evaluation. Sending the tree out to a worker would add additional communication costs to the calculation. It should also be pointed out that when trees are sent out to workers, following insertion of the 4th taxon, the master does not itself evaluate a tree. This would not be efficient as the master is usually busy with generating, sending, or receiving trees while the workers perform the evaluations.

Also shown in Figure 4.3 below are three calls to MPI_Wtime(). They correspond to the start of the insertion phase, the start of global rearrangements (and the end of the insertion phase), and the end of all tree generation and evaluation (and the end of global rearrangements). The insertion phase refers to the generation and evaluation of trees containing less than the full number of taxa, and includes local, or partial, rearrangements. Global rearrangements do not begin until all taxa have been inserted into the tree. Since the number of branches to cross during global rearrangements is generally selected by the user to be higher than that for partial rearrangements, there are many more potential topologies

generated on global rearrangements. Recall that rearrangements are performed by crossing up to and including the selected number of branches. The two phases were timed individually to gain an idea of the difference in execution time caused by what is often a significant difference in the number of trees evaluated by the two phases. The total time taken for all tree generations and evaluations is then the sum of the times for the two phases. The initializations and handling of the results were not included in the timing, as these were not the target of parallelization. The purpose of the timing was to gain an idea of the improvement in execution time, and thus performance, resulting from parallelization. In order to provide a comparison with sequential execution, the parallel program was timed while executing on only one processor. The execution time of the original, unmodified fastDNAml program would have also provided an interesting comparison, but it is not set up to use the MPI_Wtime() utility, as MPI routines are specific to parallelization.

The workers, or slaves, do not call makeDenovoTree(), but rather execute the loop in waitForWork(), shown in Figure 4.4 below. One pass through the while() loop is taken for each tree awaiting evaluation that is received from the master. For each tree, the slaveEvalTree() routine performs the branch length optimization and likelihood calculation before sending the results back to the master. A worker must probe each incoming message using MPI_Probe() to first determine the message tag, indicating the type of message. The two types of messages received by a worker have either TREE_TO_EVAL or QUIT_SLAVE for a tag. TREE_TO_EVAL signifies an incoming string representing a tree to evaluate. A QUIT_SLAVE message is received only once and causes an exit from the loop in waitForWork(), leading to termination of the worker process. The master sends

the QUIT_SLAVE message to each worker only after all tree evaluations have completed and the final program results obtained.

Each call to MPI_Recv() and MPI_Send() includes several arguments, including the message body, source or destination rank, and message tag. In the pseudo-code provided, ‘treestr’ denotes a tree’s corresponding string, and is followed by the rank of the source or destination, and the message tag. To receive a message from an unspecified source, as in recvTrees(), the argument MPI_ANY_SOURCE is used. The corresponding MPI_Recv() of an MPI_Send() call will have a matching message tag.

```
makeDenovoTree() {
  MPI_Wtime()          //time start of insertion phase
  buildSimpleTree()    //build initial tree of 3 taxa
  while( more species to add ) {
    buildNewTip()      //prep next taxon to add
    resetBestTree()    //clear list of best trees
    nextSlave = 0      //reset so start sending to first worker
    addTraverse()      //eval all (2i-5) insertion points (trees)
    recvTrees()        //recv all trees eval'd this round
    recallBestTree()   //get best resulting tree
    if( all species added ) { MPI_Wtime() }  //time start of global rearrangements
    optimize()         //rearrangements (partial or global)
  } //end while
  MPI_Wtime()          //time end of global rearrangements
  showBestTrees()      //output results
  cmpBestTrees()       //compare multiple best trees (if > 1 best tree)
} //end makeDenovoTree
```

```
testInsert(node p, node q) {
  insert(p,q)  //insert p at branch q:q->back, init br lengths, cond likelihood at p
  sendTreeforEval()   //send tree to next worker
  hookup()            //remove p to restore orig tree
  restoreZ()          //restore orig br lengths
} //end testInsert
```

```
optimize() {
  do {
    nextSlave = 0      //reset so start sending to first worker
    startOpt()         //remember starting tree
    rearrange()        //one round of branch swapping
    recvTrees()        //recv all trees eval'd this round
    recallBestTree()  //get best resulting tree to use for next round
  } while(best tree is not starting tree)  //continue while likelihood improving
} //end optimize
```

```
sendTreeforEval() {
  nextSlave++             //send to next worker
  if (nextSlave > # of workers) { //all workers sent trees
    nextSlave = 1         //so start sending to first worker again
  }
  treeString(treestr)     //convert tree struct to string
  MPI_Send(treestr, nextSlave, TREE_TO_EVAL)  //send tree
} //end sendTreeforEval
```

```
recvTrees() {
  for (all trees eval'd this round) {  //trees recv'd in any order
    MPI_Recv(treestr, MPI_ANY_SOURCE, TREE_RESULT)  //recv next tree
    str_treeReadLen(treestr)          //convert string to tree struct
    saveBestTree()                    //save eval'd tree to list of best trees if worthy
  }
} //end recvTrees
```

Figure 4.3: Pseudo-code used by the master (rank = 0) to achieve parallelization with no optimizations.

```
waitForWork() {
  MPI_Probe(tag)        //get tag of incoming msg
  while (tag == TREE_TO_EVAL) { //msg is tree to eval
    MPI_Recv(treestr, 0, TREE_TO_EVAL) //get tree string from master
    str_treeReadLen(treestr)  //convert string to tree struct
    slaveEvalTree()       //eval and return to master
    MPI_Probe(tag)        //get tag of next incoming msg
  } //end while
  //exiting while() assumes tag = QUIT_SLAVE
} // end waitForWork
```

```
slaveEvalTree() {
  smoothTree()        //branch length optimization
  evaluate()          //calc likelihood of tree
  treeString(treestr)  //convert tree struct to string
  MPI_Send(treestr, 0, TREE_RESULT)  //send result to master
} //end slaveEvalTree
```

Figure 4.4: Pseudo-code used by workers (rank > 0) to achieve parallelization with no optimizations.

4.4 Difficulties Encountered

Once implementation of the parallelization was complete and the program was executed, several issues arose that needed addressing. Adjustments were required in order to satisfy the goal of producing the same program results as fastDNAml. Certain conditions were also discovered which prevented the termination of rearrangements at the appropriate time.

4.4.1 Achieving Results Identical to fastDNAml

Additional precision was needed in the message string format in order to obtain results identical to fastDNAml. The original format of fastDNAml (and pfastDNAml) used less precision when representing likelihood and branch lengths in message strings than that shown in Figure 4.2, which was produced by the program presented here. Specifically, only 13 significant digits of the likelihood and 6 decimal places for each branch length were included in the strings. Initially this same precision was used for this work, but it was found that the numerical results differed slightly from that of original fastDNAml. For instance, -8028.45730 rather than -8028.45683 was found for the natural log of the likelihood for a specific tree. Recall that internally to the program likelihoods are represented by their natural logs, which also applies to their string representations. As the stated goal of this work was to create a parallel program yielding the same results as fastDNAml, this

difference in likelihood, although small, was not tolerable. The precision of both the likelihood and branch lengths was therefore increased to that shown in Figure 4.2. Doing so improved the results such that even after 183,000 trees were searched, the results of the two programs were identical. Increasing the precision did not appear to slow down the execution of the program.

Also found to cause differing results between this program and fastDNAml, both numerically and in the topology, was a block of code in the insert() routine called by the master. The insert() routine is called from testInsert() to generate a new topology. In fastDNAml initial estimates are calculated for the new branch lengths at the insertion point based on the Newton-Raphson method prior to optimization of the entire tree. With both the p4 parallelization and pfastDNAml, however, the master gives a much rougher estimate for these lengths using the square root of the length of the broken branch. The workers then perform the branch length optimization with four times the number of passes through the tree, as compared to fastDNAml, to make up for these poor initial estimates. Originally the parallelization carried out for this work took this same approach of having workers perform additional passes in place of the master calculating better initial values for the localized branch lengths. It was assumed that doing so would improve performance by allocating more of the computations to the workers rather than the master. Unfortunately, results comparable to fastDNAml were not achieved using this design. Perhaps this was because the new branches formed by the insertion needed more refined length values before the tree was optimized as a whole. Workers receive tree strings without any indication of which node was newly inserted, and are thus unable to provide this focused attention. Simply performing more passes over the tree is not a sufficient solution. While it was assumed that

a reduced performance would result from allowing the master to calculate better initial lengths for the new branches, achieving results identical to fastDNAml was more important. In this case, the workers were no longer expected to carry out the 4-fold increase in the number of passes. In the end it was discovered that this change did not have a noticeable effect on performance.

4.4.2 Preventing Infinite Rearrangements

Another problem with the initial parallelization was found during rearrangements, where under certain conditions the master was unable to properly determine that the likelihood was no longer improving, thus preventing exit from the do/while() loop in optimize(). On closer inspection the problem was found to be in the function saveBestTree() that was unmodified from fastDNAml. When the master calls saveBestTree() to determine if a newly evaluated topology should be saved in its ‘bestlist’, the likelihood is compared with that of the lowest-scoring topology in the list. The original design was such that if the new likelihood was equal to or greater than the worst likelihood in the list, then the worst-scoring topology was replaced by the new topology. This presents a problem in optimize() when the ‘bestlist’ has a size of one, where the worst topology is also the best topology. Recall that to break out of the do/while() loop in optimize() the likelihood must not improve on consecutive rounds. However, at the end of each round a comparison is not made of the likelihoods themselves, but of the topologies of the new best tree and of the starting tree, which was the best tree of the previous round. It may be the case that different topologies have equal likelihoods, especially considering the precision lost on string conversions, and in this situation a new tree would replace the starting tree as the best tree. The program then behaves as if the likelihood improved, continuing with another round of

rearrangements. Since performing rearrangements on a tree that was produced from rearrangements on a given starting tree may again produce that original starting tree, and all may have equal likelihoods, the situation could arise where the condition to break out of the do/while() loop is never met. To avoid this, saveBestTree() was modified so that new topologies are only saved to the 'bestlist' when their likelihood is greater than that of the worst topology in the list, and topologies with the same likelihood as the worst are discarded. This brings to light the recommendation that the user select a size for the 'bestlist' greater than 1, such as 10 or 15. With fewer topologies saved in the 'bestlist' there is a greater chance that top topologies with equal likelihoods will be lost. It should also be noted that topologies in 'bestlist' with equal likelihoods may appear in different orders on different runs of the same program conditions, with the order dependent on when evaluated trees are received by the master.

4.5 Comparison with pfastDNAml

The pfastDNAml program [38] is considered to be the 'official' parallelization of fastDNAml and as such uses fastDNAml for a foundation, as does the parallelization presented in this work. Thus, a comparison is warranted. Both programs utilize the master/workers scheme, with parallelization achieved by workers evaluating individual trees in parallel. Both programs incorporate this approach into the algorithm of fastDNAml as shown in Figure 2.5. The main difference stems from the use of two processors for control in pfastDNAml, specifically a master process and a foreman process. Here the master is given the responsibility of generating all trees in each round. It is the foreman, however, that communicates with the workers. The master sends all trees needing

evaluation to the foreman, which then sends them out to the workers. Once each worker has been sent a tree, a worker does not receive another tree until the foreman receives its results from the previous tree. The foreman also keeps track of which trees are sent to which workers, and when a result is not received within a specified amount of time the associated worker no longer receives trees and the tree is sent to a different worker. The foreman uses a 'work queue' and a 'ready queue' to accomplish its tasks. After each round, the foreman sends the master the top likelihood tree determined from the results. An optional monitor process is also included that displays the progress as the program executes. Worker processes are given the same responsibilities in both programs, which is to calculate the branch lengths and likelihood of a tree.

Many other differences in implementation exist between the two programs, as the only code from pfastDNAml that was incorporated into this work was related to initializations and string representation of trees. Most of these differences, however, are insignificant. One interesting difference that should be noted is that pfastDNAml provides PVM (parallel virtual machine), MPI, and serial implementations, with the program code using generic calls to keep it independent of the selected library. Also of note is that in Figure 2.5, which was presented in a publication describing pfastDNAml, evaluation of the 3-taxon tree is shown to be performed by a worker, but upon inspection of the pfastDNAml code it appears as if the master instead evaluates this tree. Another oddity is that pfastDNAml offers the quickadd feature, the 'Q' option, but the code implies that this has no effect in parallel execution. The pfastDNAml code also suggests that the results produced are not identical to fastDNAml, when considering the adjustments needed for this work. This was not verified, however, due to time constraints. Consideration was given to using

pfastDNAml rather than sequential fastDNAml as a foundation, but as the focus of this work was parallelization, specifically optimization of it, it was felt that a better understanding would be gained through implementation. Also, the traditional master/workers scheme, lacking a foreman, was desired to facilitate adaptation to other programs for relief of the associated bottleneck.

fastDNAml was successfully parallelized following the master/workers technique as discussed in this chapter, fulfilling the first major goal of this work. Several adjustments made in order to arrive at results identical to those produced by fastDNAml were also described. While the work presented thus far executes the algorithm of fastDNAml in a significantly reduced amount of time, it suffers from a communication bottleneck, limiting the improvement in performance as the system and problem size increase. It is the highlight of this research to alleviate this bottleneck through the three optimizations presented in the next chapter.

Chapter 5 Optimizations to Parallelization

The parallelization described in the previous chapter suffers from the communication bottleneck typical of the master/workers approach. Through the introduction of three optimizations, specifically message packing, workers keeping only the best trees, and multiple masters, the hope is that this bottleneck will be reduced, making analysis with large-scale systems and problems feasible. These optimizations form the core of this thesis work, and the design and implementation of each is provided in this chapter, along with details on how they are expected to lead to relief of the bottleneck.

5.1 Message Packing

Message packing is performed by the master and involves the combination of multiple trees awaiting evaluation into a single message, or bundle, rather than sending each tree out individually. The purpose of message packing is to reduce message overheads at the master by decreasing the number of outgoing messages, while the total number of trees sent to the workers for evaluation remains unchanged. The number of trees constituting a bundle is selected by the user, where a bundle is then a text string formed by concatenating the representative text strings of the corresponding number of trees, as shown in Figure 5.4. As topologies are generated, those ready for evaluation are stored by the master in a partially complete bundle, which is then sent out to a worker once it contains the specified number of trees. Thus consecutive topologies are no longer dispersed over different workers, but rather a single worker receives a group of consecutive topologies. Recall, however, that the order in which topologies are evaluated does not affect the results. Upon receiving a

bundle, a worker parses out one tree at a time from the string, which it then evaluates and returns to the master as an individual message, as before.

Figure 5.1 below depicts how parallelization is carried out using message packing. This should be compared with Figure 4.1 showing parallelization in the absence of message packing. A round in which 14 topologies are generated is used for the example in Figure 5.1, where four trees have been selected as the bundle size. The small red arrows originating and terminating at the master each represent the generation of a topology, which is stored in the bundle being assembled. Once full, the bundle is sent off to the next worker, shown with red arrows that contain blocks representing bundles. Notice that the last bundle contains only two trees. Because four does not evenly divide into 14, the master sends out a smaller bundle containing the remaining trees before receiving the results. The process of returning the results, here involving 14 messages for the 14 trees, is not affected by message packing.

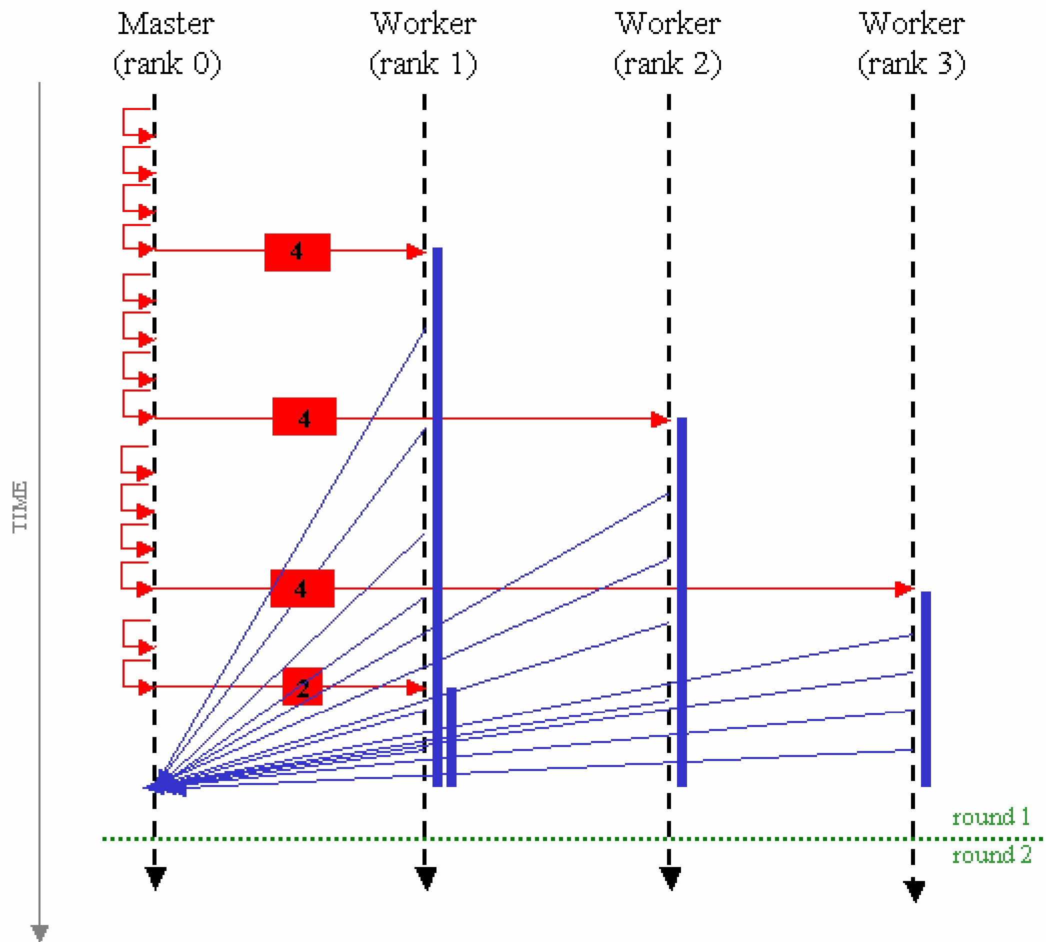


Figure 5.1: Sequence of sending and receiving trees between processors for one round of evaluations using message packing. Four trees constitute a bundle and 14 trees are evaluated in the round. Compare to Figure 4.1.

5.1.1 Implementation

Message packing was incorporated into the parallelization described in Chapter 4 as an option that can be turned off by the user through the use of a switch. The associated pseudo-code for the master and worker processes is provided in Figures 5.2 and 5.3 below. The switch is not shown and is assumed to be turned on. Only those routines that were modified or added to achieve message packing have been included. The initializations

remain unchanged from Chapter 4, with the addition of dynamic memory allocation for a string to hold the incoming and outgoing bundle strings, needed by all processors. The string size allocated is based on the selected number of trees per bundle as well as the maximum tree size. Previously, allocation was only needed for a string representing one tree.

```
makeDenovoTree() {
  MPI_Wtime()          //time start of insertion phase
  buildSimpleTree()    //build initial tree of 3 taxa
  while( more species to add ) {
    buildNewTip()      //prep next taxon to add
    resetBestTree()    //clear list of best trees
    nextSlave = 0      //reset so start sending to first worker
    addTraverse()      //eval all (2i-5) insertion points (trees)
    packMsg(tr, FALSE) //send off any remaining trees
    recvTrees()        //recv all trees eval'd this round
    recallBestTree()   //get best resulting tree
    if( all species added ) { MPI_Wtime() }  //time start of global rearrangements
    optimize()         //rearrangements (partial or global)
  } //end while
  MPI_Wtime()          //time end of global rearrangements
  showBestTrees()      //output results
  cmpBestTrees()       //compare multiple best trees (if > 1 best tree)
} //end makeDenovoTree
```

```
optimize() {
  do {
    nextSlave = 0        //reset so start sending to first worker
    startOpt()           //remember starting tree
    rearrange()          //one round of branch swapping
    packMsg(tr, FALSE)   //send off any remaining trees
    recvTrees()          //recv all trees eval'd this round
    recallBestTree()     //get best resulting tree to use for next round
  } while(best tree is not starting tree)  //continue while likelihood improving
} //end optimize
```

```
sendTreeforEval() {
  packMsg(tr, TRUE)      //add new tree to bundle and send bundle if now full
} //end sendTreeforEval
```

```
packMsg(tr, moreTrees) {
  if(moreTrees) {  //another new tr needs to be sent out
    treeString(treestr)  //convert tree struct to string
    add new tr to existing bundle string
   }
  if(bundle full || (!moreTrees & trees in bundle)) { //send bundle
     nextSlave++            //send to next worker
     if (nextSlave > # of workers) { //all workers sent trees
       nextSlave = 1        //so start sending to first worker again
     }
     if(1 tree in bundle) { //send tr individually (no bundle)
       MPI_Send(bundlestr, nextSlave, TREE_TO_EVAL)
     } else { //send whole bundle
       attach number of trees in bundle to end of string
       MPI_Send(bundlestr, nextSlave, BUNDLE_TO_EVAL)
     }
   }//end if (bundle full…)
}//end packMsg
```

Figure 5.2: Pseudo-code used by the master to achieve parallelization with message packing. Only routines that were modified for message packing are provided.

```
waitForWork() {
  MPI_Probe(tag)                //get tag of incoming msg
  while (tag == TREE_TO_EVAL or BUNDLE_TO_EVAL) {
    if(tag == TREE_TO_EVAL) {
      MPI_Recv(treestr, 0, TREE_TO_EVAL)  //get tree string from master
      str_treeReadLen(treestr) //convert string to tree struct
      slaveEvalTree()          //eval and return to master
    } else { // BUNDLE_TO_EVAL
      MPI_Recv(bundlestr, 0, BUNDLE_TO_EVAL)  //get bundle string from master
      unpackBundle()            //get trees in bundle and evaluate
    }
    MPI_Probe(tag)              //get tag of next incoming msg
  } //end while
    //exiting while() assumes tag = QUIT_SLAVE
} //end waitForWork
```

```
unpackBundle() {
  get number of tree strings in bundle from end of bundle string
  while (still trees in bundle) {
    take next tree string from beginning of bundle string
    str_treeReadLen(treestr)     //convert string to tree struct
    slaveEvalTree()              //eval tree and return to master
  } //end while
} //end unpackBundle
```

Figure 5.3: Pseudo-code used by workers to achieve parallelization with message packing. Only routines that were modified for message packing are provided.

Comparing the makeDenovoTree() and optimize() routines executed by the master with and without message packing, in Figures 5.2 and 4.3, respectively, it can be seen that the only difference is the addition of a call to the new routine packMsg(), whose pseudo-code is also provided. When executing makeDenovoTree(), as well as optimize(), packMsg() is called from two locations on each round. In a given round, packMsg() is first called from sendTreeforEval() for each topology generated with addTraverse(). packMsg() replaces the body of sendTreeforEval(), converting the transmission of individual trees into a bundling process. Two arguments are passed to packMsg(), with the second being a flag indicating if the first argument is a new tree to be bundled and eventually sent out to a worker. When this is not true, the call to packMsg() indicates that the tree generation process has completed, and that any topologies waiting in a partially-filled bundle should be sent as a smaller bundle. This is the case in makeDenovoTree() and optimize() when packMsg() is just before receiving results.

A bundle string containing three trees as produced by packMsg() is given below in Figure 5.4. The last character in the bundle indicates the number of trees in the bundle. The newline is placed after each tree string by the treeString() routine from pfastDNAml. When

a bundle is sent out to a worker, the master uses the message tag BUNDLE_TO_EVAL, as opposed to TREE_TO_EVAL, which alerts the worker that a bundle needs to be handled. When there is only one tree remaining to form a bundle following completion of a round of topology generations, rather than send a bundle containing one topology it is sent as an individual tree message with the TREE_TO_EVAL tag.

```
[&&fastDNAml: version = '1.2.2', likelihood = -
1.00000000000000052504760255e+300, ntaxa = 5, opt_level = 0, smoothed
= 1] ('Sequence5': 0.027289047263990,('Sequence4':
0.144262134601984,('Sequence2': 0.045422338265723,'Sequence3':
0.070091754201706): 0.057929540285774): 0.000000583212366,'Sequence1':
0.103620433740958):0.0;
[&&fastDNAml: version = '1.2.2', likelihood = -
1.00000000000000052504760255e+300, ntaxa = 5, opt_level = 0, smoothed
= 1] (('Sequence5': 0.023626156784827,'Sequence4': 0.062451487921840):
0.081810656386423,('Sequence2': 0.045422338265723,'Sequence3':
0.070091754201706): 0.057929540285774,'Sequence1':
0.103620274121730):0.0;
[&&fastDNAml: version = '1.2.2', likelihood = -
1.00000000000000052504760255e+300, ntaxa = 5, opt_level = 0, smoothed
= 1] ('Sequence4': 0.144262134601984,('Sequence5':
0.016570877799449,('Sequence2': 0.045422338265723,'Sequence3':
0.070091754201706): 0.057929541615129): 0.000000583212366,'Sequence1':
0.103620274121730):0.0;
3
```

Figure 5.4: Example message bundle string containing 3 tree strings, each denoted by a different color. The number of trees in the bundle is indicated by the last character shown in the bundle string. Note that each tree string is terminated by a newline character.

Figure 5.3 shows the modifications to the code for message packing as executed by the worker processes. The workers are unaware when the message packing option is in effect, instead relying on the tag of the incoming message to determine what actions to take. The case for BUNDLE_TO_EVAL was added to the waitForWork() loop, which results in a call to unpackBundle(). As shown in Figure 5.3, this routine, added for message packing, first reads the number of trees in the bundle, then removes each tree in the bundle one at a

time. Immediately thereafter, before removing the next tree from the bundle, the tree is evaluated and the associated results are sent off to the master. This is accomplished through a call to slaveEvalTree(), which needed no modification for message packing. The communication of tree results, as mentioned previously, did not change with the addition of message packing, as the master waits to receive, in any order, a number of messages equal to the number of trees generated and sent out to the workers.

When the program was initially run after adding the message packing option, an error caused abnormal termination. A message was output recommending that an increase be made in the environment variable P4_GLOBMEMSIZE to gain more memory. With an input file containing 50 taxa and 3 trees per bundle, runs using up to 30 processors successfully completed, with the error showing up with 40 or more processors. However, on increasing the bundle size to 5 the problem appeared earlier at 30 or more processors. In all cases the crash occurred during global rearrangements, where large numbers of trees are sent out in each round (8,742 in this case). The cause was narrowed down to a call to MPI_Send() in packMsg() where the bundle string is sent to a worker. Increasing the P4_GLOBMEMSIZE value from 4194304 bytes to 264241152 bytes solved the problem under the conditions run for this work, using up to the available 94 processors. However it is feared that the error may reappear once larger datasets, bundle sizes and/or numbers of processors are selected. The exact cause of the problem was never determined, though it could be related to exceeding the limits of a possibly finite-sized outgoing message buffer used by the asynchronous, blocking MPI_Send().

5.1.2 Performance Considerations

Message packing was introduced as a means to alleviate the communication bottleneck caused by a single master handling many workers and trees. Specifically, combining several short messages into one long message cuts down on overheads faced by the master that are independent of the size of the message. Overheads require processing and may also lengthen communication time, thus adding to the execution time and limiting performance. The overheads associated with each message may result from a number of sources external to the programmer's view, including initiation of the transfer, addition of headers to the message, etc. Sources of overheads are also evident from inspection of the code, such as the arguments to MPI_Send() for source, destination, tag, etc., as well as the function call itself. It is not the purpose of this research to pinpoint or analyze the overheads, but rather to reduce the execution time resulting from these overheads by decreasing the number of messages sent out by the master. With less time dedicated to overheads, the master can more efficiently deal with the demands of its workers, thus easing the bottleneck.

There are several ways in which message packing could potentially lead to a reduction in performance, but it is expected that this will be outweighed by the performance gained through reduced overheads. One instance where message packing may negatively affect performance occurs at the beginning of each round of topology generation, where workers sit idle while trees needing evaluation wait at the master until enough topologies are generated to fill a bundle. The larger the bundle size, the more time is wasted before workers receive topologies to evaluate. While the generation of a topology is not as time-consuming as compared with calculation of its branch lengths and likelihood, a considerable amount of time is still required. As more trees are produced as a round

progresses, and all workers have received bundles, the master is expected to be able to keep up with the workers' demands for work, and improvement in performance should be apparent.

An unbalanced workload over the workers caused by message packing may also hinder performance. As the bundle size increases, so does the difference in the number of trees assigned to each of workers for evaluation. In the absence of message packing, this difference is at most one tree. Consider an extreme example with three trees generated in a round and a bundle size of three. Once full, the bundle is sent to the first of many workers, and more time is needed for that worker process to evaluate the three trees sequentially than if each tree were sent to a different worker, allowing parallel computation. Thus, it is evident that the number of trees constituting a bundle is an important factor in message packing, with larger bundles more detrimental to performance in some cases, but also resulting in fewer message overheads. As higher numbers of trees are generated per round, the benefits of reduced overheads should become more noticeable. However, while overheads at the master associated with sending a message are reduced, overheads for managing the bundles arise, with the effects scaling with number of trees.

5.2 Workers Keep Best Trees

In the parallelization presented earlier in Chapter 4, workers return their results to the master immediately after a tree is evaluated. The option was added where workers keep the best trees evaluated, accomplished through the use of a 'bestlist' structure allocated to each worker. This structure follows the same design as that of the master described in section 3.2.2, where trees with the best likelihood scores are saved to the list and the lower-scoring

trees are discarded. The number of trees capable of being stored in each worker's 'bestlist' is the same as that for the master, and is selected by the user. Note that the size of the 'bestlist's is not specific to this optimization, and is determined from the 'K' user option described in Table 3.1. At the end of each round, the master commands each worker that received trees in that round to send the top-scoring trees' results in its 'bestlist' as one message bundle. The bundles are received in any order. When receiving each of the bundles, which have the same format as in Figure 5.4, the master parses out each tree, saving it to its own 'bestlist' if worthy. Most of the trees received from the workers will likely be discarded, as the master must consider all trees received from all workers. The elimination of trees by the workers does not affect the resulting trees in the master's 'bestlist', as those top trees are guaranteed to have been saved among the workers. The trees discarded by the workers would have otherwise been discarded by the master, thus the program results are unchanged. This design can lead to a significant reduction in the number of trees received by the master, decreasing the amount of work required at the end of each round. As a result, the workers spend less time idle before the master is once again able to generate topologies from the newly determined starting tree.

Figure 5.5 below shows how parallelization is carried out when each worker is given a 'bestlist.' Eight trees are evaluated in the round, with the top two trees stored in each 'bestlist', as selected by the user. The generation and transmission of individual trees to the workers for evaluation is unchanged. Note here that the message packing option is turned off. The workers receive and evaluate each tree as before, but now rather than send off the results, the tree is saved in the 'bestlist' following evaluation if its likelihood is determined to be among the two best calculated thus far. The green arrows represent a message sent by

the master to each worker requesting the return of the results in each ‘bestlist’, indicating the end of the round. Only workers which received trees in the round are sent the request. Thus, if there were nine workers rather than the three shown, eight bundles would be received by the master. Also of note in this situation is that each ‘bestlist’ would not contain a full number of trees, although the results would still be returned as a bundle. When a ‘bestlist’ containing only one tree result is to be returned, a bundle of one tree is sent. This is opposed to sending the tree as an individual tree message, which would use a different message tag and not have the number of trees in the bundle appended.

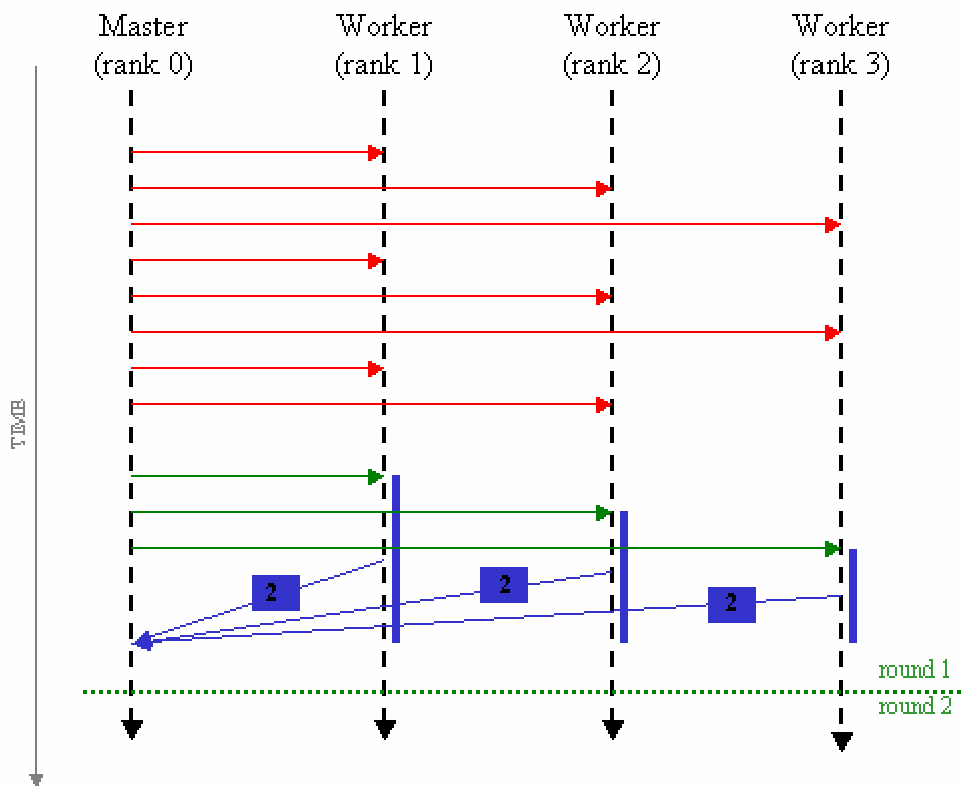


Figure 5.5: Sequence of sending and receiving trees between processors for one round of evaluations when workers keep their best trees. Here, eight trees are evaluated in the round, and the top two trees of each worker are held in its ‘bestlist’. The workers return their top trees to the master as a bundle when commanded at the end of the round. The green arrows represent the message sent by the master to each worker prompting them to return their ‘bestlist’s. Compare to Figure 4.1.

The bundling of trees from each worker's 'bestlist' should not be confused with message packing described in the previous section, which is a separate option. Although the same bundle format in Figure 5.4 is used for both options, message packing only affects bundling by the master of outgoing trees needing evaluation, not the return of results.

5.2.1 Implementation

Similar to message packing, this option was incorporated into the parallelization using a switch. Figures 5.6 and 5.7 below display the pseudo-code for the master and workers, respectively, when each worker maintains a 'bestlist'. The switch is not shown, nor is pseudo-code specific to other options, such as message packing. Initializations are similar to before, but now allocation and initialization of the 'bestlist' structure is performed by all processes, rather than just the master. Larger strings are also allocated to handle the bundling of the results. Additional routines were not added when implementing this option, however unpackBundle() was reused from the message packing option. This routine was modified only slightly by replacing the call to slaveEvalTree() with saveBestTree(), since the master, rather than the workers, receives the bundles in this situation. The packMsg() routine from message packing was not reused, however, as here workers assemble the bundle all at once from pre-existing trees, rather than maintaining a partial bundle until a sufficient number of trees become available. A new message tag of BUNDLE_RESULT is also used rather than BUNDLE_TO_EVAL. For this option, the code executed to assemble bundles was put directly into waitForWork(). Specifically, the bundling is performed in the newly added case for the SEND_BEST_TREES tag of the incoming message. Such a message is sent to each worker indicating the end of the round when the 'bestlist's are to be bundled and sent to the master. Workers' 'bestlist's are then cleared in preparation for the

start of the next round. The only other major change to the workers' code was in slaveEvalTree(), where saveBestTree() is called to save the newly evaluated tree to the 'bestlist' rather than sending it off. The option switch, though not shown, would be used in this situation to choose between the two actions.

Functionality was added to the makeDenovoTree(), optimize() and sendTreeforEval() routines called by the master to store the rank of the highest ranking worker that received trees in each round. The master then knows to only request 'bestlist's from those workers. The receipt of results is accomplished through a call to recvTrees(), as before, which is now specialized to handle incoming bundles. The messages requesting results are sent out to the workers before any calls are made to receive the bundles. The order in which the evaluated trees appear in the 'bestlist' bundle is insignificant, as once a tree is parsed out only the likelihood is needed to determine the appropriate location, if any, in the master's 'bestlist'.

```
makeDenovoTree() {
  MPI_Wtime()          //time start of insertion phase
  buildSimpleTree()    //build initial tree of 3 taxa
  while( more species to add ) {
    buildNewTip()      //prep next taxon to add
    resetBestTree()    //clear list of best trees
    maxSlaveSent  = nextSlave = 0  //reset so start sending to first worker
    addTraverse()      //eval all (2i-5) insertion points (trees)
    recvTrees()        //recv all trees eval'd this round
    recallBestTree()   //get best resulting tree
    if( all species added ) { MPI_Wtime() }  //time start of global rearrangements
    optimize()         //rearrangements (partial or global)
  } //end while
  MPI_Wtime()         //time end of global rearrangements
  showBestTrees()      //output results
  cmpBestTrees()       //compare multiple best trees (if > 1 best tree)
} //end makeDenovoTree
```

```
optimize() {
  do {
    maxSlaveSent = nextSlave = 0    //reset so start sending to first worker
    startOpt()           //remember starting tree
    rearrange()        //one round of branch swapping
    recvTrees()        //recv all trees eval'd this round
    recallBestTree()  //get best resulting tree to use for next round
  } while(best tree is not starting tree)  //continue while likelihood improving
} //end optimize


sendTreeforEval() {
  nextSlave++             //send to next worker
  if (nextSlave > # of workers) { //all workers sent trees
    nextSlave = 1        //so start sending to first worker again
  }
  //remember max worker that recv'd trees this round
  maxSlaveSent = MAX(maxSlaveSent, nextSlave)
  treeString(treestr)      //convert tree struct to string
  MPI_Send(treestr, nextSlave, TREE_TO_EVAL)  //send tree
} //end sendTreeforEval


recvTrees() {
  for(each worker that received trees this round) {
    MPI_Send( dummy string, worker, SEND_BEST_TREES) //request bestlists
  }
  for(# of workers that received trees this round) {
   MPI_Recv( bundlestr,MPI_ANY_SOURCE, BUNDLE_RESULT) //recv in any order
   unpackBundle(bundlestr) //get trees in bundle and save to bestlist
  }
} //end recvTrees


unpackBundle() {
  get # of tree strings in bundle from end of bundle string
  while(still trees in bundle) {
    take next tree string at beginning of bundle string
    str_treeReadLen(treestr)   //convert string to tree struct
    saveBestTree()              //save tree to bestlist if worthy
  }
} //end unpackBundle
```

Figure 5.6: Pseudo-code used by the master to achieve parallelization when workers keep their best trees. Only routines that were modified for this option are provided.

```
waitForWork() {
  MPI_Probe(tag)                 //get tag of incoming msg
  while (tag == TREE_TO_EVAL or SEND_BEST_TREES) { //msg is tree to eval
    if(tag == TREE_TO_EVAL) {
      MPI_Recv(treestr, 0, TREE_TO_EVAL) //get tree string from master
      str_treeReadLen(treestr)  //convert string to tree struct
      slaveEvalTree()            //eval and return to master
    } else { // SEND_BEST_TREES
      for (each topol in bestlist) {
        treeString()               //convert tree struct to string
        add tree to bundle
      } //end for
      attach number of trees to end of bundle
      MPI_Send( bundleStr, 0, BUNDLE_RESULT)
      resetBestTree()              //clear bestlist
    } //END SEND_BEST_TREES
    MPI_Probe(tag)               //get tag of next incoming msg
  } //end while
  //exiting while() assumes tag = QUIT_SLAVE
} //end waitForWork


slaveEvalTree() {
  smoothTree()        //branch length optimization
  evaluate()          //calc likelihood of tree
  saveBestTree()      //save tree to bestlist if worthy
} //end slaveEvalTree
```

Figure 5.7: Pseudo-code used by workers to achieve parallelization when keeping their best trees. Only routines that were modified for this option are provided.

5.2.2 Performance Considerations

The performance enhancement potentially gained from this option is evident. A significant reduction in the number of evaluated trees sent to the master reduces the overall execution time as less time is needed for the master to determine the top-scoring trees from each round while the workers sit idle. The communication bottleneck is most prominent at this point, following completion of a round of topology generation, since in the absence of this

option the master must handle the results from all trees evaluated in the round. With only a single master process utilized, each tree must be received and examined sequentially. This reduces the efficiency of parallelization as all worker processors remain idle while the single master processor is executing.

Several factors affect the extent by which performance improves. Most significant, perhaps, is the number of trees requiring evaluation in a given round. Initially, at the start of the tree building process, a smaller number of trees are generated per round, which is likely not sufficient to fill the 'bestlist's. Trees are therefore not discarded by the workers. No reduction is seen in the number of trees received by the master, which must perform additional processing with the optimization, including the request for results and parsing of bundles. There is a chance that with fewer messages received due to bundling, a reduction in the overhead associated with receiving messages may result, consequently improving performance. However, significantly time-consuming overheads are more often associated with sending messages. Thus, when a small number of trees are generated in a round, a reduction, rather than improvement, in performance may arise. Once the program progresses to the point where sufficient numbers of trees are generated such that trees are discarded, the amount by which performance improves is expected to increases with the number of trees.

The number of workers can also have an impact on performance when this option is in effect. With fewer workers, the 'bestlist's fill up sooner, causing more trees to be discarded. Thus, less trees are received by the master, increasing performance. On a similar note, overheads of receiving bundles are reduced since there are fewer workers requiring

requests for results, and fewer bundles. However, the performance gained through parallelization improves dramatically as the number of workers increases, and any reduced improvement in performance associated with this option will likely be overshadowed, assuming a sufficient number of trees are generated. The size selected for the 'bestlist' affects performance in a manner similar to the number of workers, in that smaller sizes result in more rapid filling of the 'bestlist's and subsequent discarding of trees. One final note concerning this option is that the overheads associated with maintaining a 'bestlist' are likely to slow each worker down slightly, but as the workers operate in parallel the effect should be minimal, especially when considering the substantial improvement in the bottleneck at the master.

5.3 Multiple Masters

As discussed on several previous occasions, the performance gained from parallelization with the master/workers approach is limited as the system and problem sizes increase. This mainly results from the use of a single master process that must generate work, specifically topologies, for all workers and manage all incoming results. With an increasing number of trees, one would expect additional workers to offset the effects by performing the evaluations in parallel. However, the master also faces an increased workload, which it must handle sequentially, thus limiting the potential performance gain. Allowing the master's responsibilities to be executed in parallel would alleviate this bottleneck. This is the aim of the multiple masters optimization. Specifically, parallelization of the master's tasks is accomplished via the use of multiple 'extra' master processes that operate in parallel, and are managed by a single 'head' master process. The head master

communicates only with the extra masters, each of which communicates with a subset of workers assigned to that extra master for the entire program duration. On a given round, each extra master is responsible for generating a subset of the topologies, sending them out to its assigned workers for evaluation, and receiving the associated results from those workers. Once the results are received, with the top-scoring trees saved to a 'bestlist', the extra masters send their bundled 'bestlist's to the head master. The head master then treats the incoming bundles in the same manner as previously described when workers keep their best trees, where each tree removed from the bundle is either saved to the head master's 'bestlist' or discarded. The top tree received over all extra masters is then used as the starting tree for the next round. The top tree found after every round is not changed by the addition of multiple masters. The head master sends the starting tree to all extra masters in order that they all begin topology generation from the correct starting tree on the next round. The initial 3-taxa starting tree is created by all extra masters, rather than received from the head master. Figure 5.8 shows the organization of processors for an example with 12 total processors, three of which are designated by the user as extra masters. Rank 0 corresponds to the head master, ranks 1 through the number of extra masters are reserved for those extra masters, with all remaining ranks assigned to worker processes. Note that for a given number of processors, fewer workers are available with multiple masters. The workers, along with topology generation, are divided evenly over the extra masters, as will be discussed in the next section.

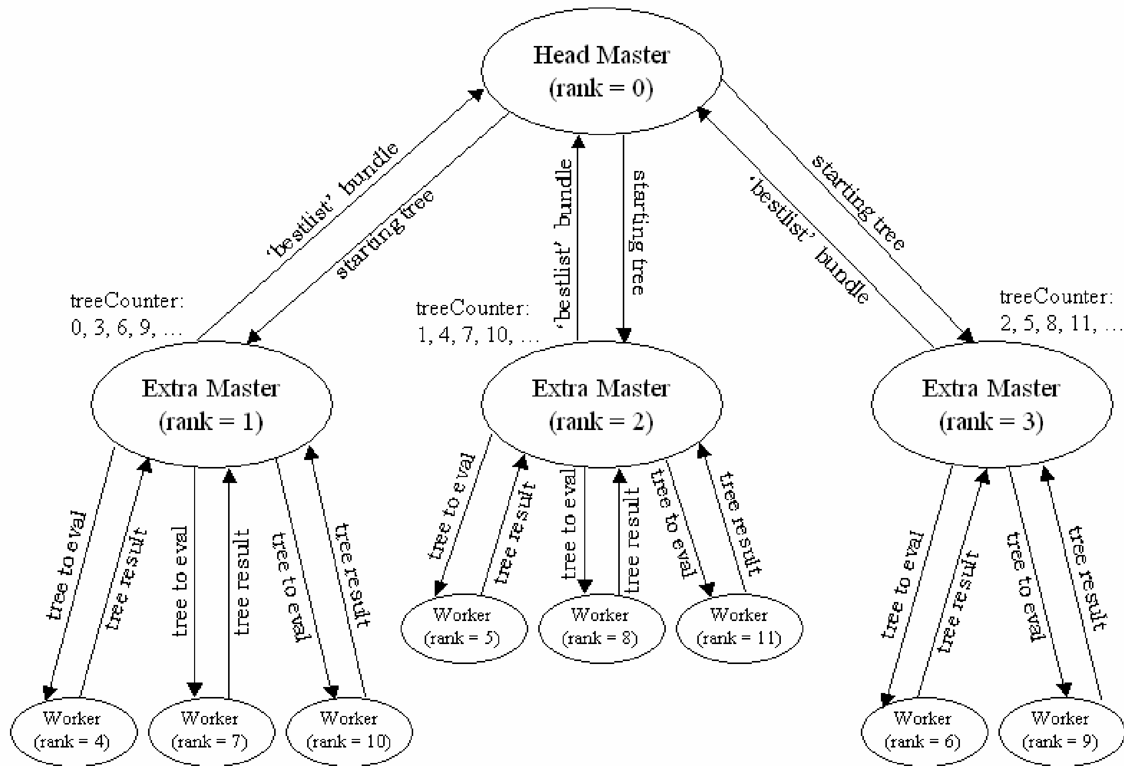


Figure 5.8: Organization of processors with multiple masters. Here, 12 total processors are utilized, with three extra masters selected. 'treeCounter' indicates which trees are generated by which extra masters in a given round.

5.3.1 Implementation

Optimization through multiple masters could be implemented in many ways, with a simpler approach taken here to demonstrate the concept. Similar to the other options previously described, the user may elect to use this option with a switch, and must select the desired number of extra masters as well. As multiple masters are only practical when enough workers exist to assign at least one worker to each extra master, the program reports an error when the user does not select an appropriate number of processors. It should also be noted that selecting one extra master will not result in a performance gain, although permitted by the program. An automatic switch was also created that allows the program to

determine the best number of extra masters, rather than the user, as described in later section. The multiple masters option was implemented so as to be compatible with the two other options previously described when selected by the user. The implementation described in this section, however, assumes these options are not in effect. Following the typical program initializations performed by all processors, the code separates into three sections corresponding to the three different roles. Workers, as before, execute the waitForWork() loop provided in Figure 5.11. The head master and extra masters execute variants of the makeDenovoTree() loop, which are substantially different from each other as well as from the original routine. Therefore, to facilitate implementation and comprehension of the final code, three separate routines are used to carry out the three separate cases. The makeDenovoTree() routine is reserved for the absence of multiple masters, while hm_makeDenovoTree() is executed by the head master, and extra masters execute em_makeDenovoTree(). Pseudo-code for these last two routines is provided in Figures 5.9 and 5.10, respectively. Both routines exit as before, following generation and evaluation of all trees in the program, leading to process termination. The head master is responsible for sending the appropriate message, with tag QUIT_SLAVE, to each worker to signal termination.

```
hm_makeDenovoTree() {
  MPI_Wtime()             //time start of insertion phase
  while( more species to add ) {
    resetBestTree()       //clear bestlist
    for( all extra masters ) { //recv in any order
      MPI_Recv( bundlestr, MPI_ANY_SOURCE, EM_BUNDLE_INSERT)
      read number of trees generated from bundle string
      if(bundle not empty) {
        unpackBundle()  //get trees in bundle and save to bestlist
      }
```

```
    } //end for
    recallBestTree()       //get best resulting tree
    treeString(treestr)    //create string from best resulting tree
    for( each extra master ) {  //send starting tree to each extra master
      MPI_Send( treestr, extra master, EM_BEST_INSERT)
    }
    if( all species added ) { MPI_Wtime() }  //time start of global rearrangements
    hm_optimize()          //rearrangements (partial or global)
  } //end while
  MPI_Wtime()              //time end of global rearrangements
  showBestTrees()          //output results
  cmpBestTrees()           //compare multiple best trees (if > 1 best tree)
} //end hm_makeDenovoTree


hm_optimize() {
  do {
    startOpt()                    //remember starting tree
      for( all extra masters ) { //recv in any order
        MPI_Recv( bundlestr, MPI_ANY_SOURCE, EM_BUNDLE_OPT )
        read number of trees generated from bundle string
         if(bundle not empty) {
           unpackBundle(0)     //get trees in bundle and save to bestlist
         }
      } //end for
      recallBestTree()            //get best resulting tree
      treeString(treestr)         //create string from best resulting tree
      for( each extra master ) { //send starting tree to each EM
        if (best tree is not starting tree) {//send starting tree since likelihood improving
          MPI_Send( treestr, extra master, EM_BEST_OPT)
        } else { //stop since likelihood not improving
        MPI_Send( treestr, extra master, EM_STOP_OPT)
        }
      } //end for
  } while(best tree is not starting tree)  //continue while likelihood improving
} //end hm_optimize
```

Figure 5.9: Pseudo-code used by the head master (rank = 0) when the multiple masters option is in effect.

```
em_makeDenovoTree() {
  buildSimpleTree()       //build initial tree of 3 taxa
  while( more species to add ) {
    buildNewTip()         //prep next taxon to add
    nextSlave = slaveRankStart + my_rank – 1  //rank of first worker to send to
    treeCounter = -1      //reset for new round of trees
    addTraverse()         //eval trees assigned to this extra master
    recvTrees()           //recv all results for trees gen'd by this extra master this round
    add number trees gen'd by this extra master to start of bundle string (may be 0)
    for (each topol in bestlist) {
      treeString(treestr)  //create string from tr
      add tree to bundle
    } //end for
    attach number of trees to end of bundle string
    MPI_Send( bundlestr, 0, EM_BUNDLE_INSERT)
    resetBestTree()               //clear bestlist
    MPI_Recv( treestr, 0, EM_BEST_INSERT)  //recv starting tree for next round
    str_treeReadLen(treestr)  //convert string to tree
    em_optimize()                //rearrangements (partial or global)
  } //end while
} //end em_makeDenovoTree


addTraverse(node p, node start) {
  //node 'p' to be inserted into tree
  //node 'start' indicates insertion location (start of recursion)
  //select 'start' carefully for traversal of entire tree
  //traversal only moves forward (i.e. not in direction of start->back)
  treeCounter++          //generating next tree
  if((treeCounter%numExtraMasters) == (my_rank-1)) { //if tree assigned to extra master
    testInsert(p, start)  //insert p and evaluate, then undo
  }
  if(p != tip) {  //continue traversing tree in forward direction
    addTraverse(p, start->next->back)            //repeat for neighboring branch
    addTraverse(p, start->next->next->back)    //repeat for neighboring branch
  } //end if
} //end addTraverse


testInsert(node p, node q) {
  insert(p,q)  //insert p at branch q:q->back, init br lengths, cond likelihood at p
  em_sendTreeforEval()  //send tree to next assigned worker
  hookup()                     //remove p to restore orig tree
  restoreZ()                   //restore orig br lengths
} //end testInsert
```

```
em_optimize() {
  do {
    nextSlave = slaveRankStart + my_rank – 1  //rank of first worker to send to
    treeCounter = -1 //reset for new round of trees
    rearrange()       //one round of branch swapping
    recvTrees()       //recv all results for trees gen'd by this extra master this round
    add number trees gen'd by this extra master to start of bundle string (may be 0)
    for (each topol in bestlist) {
      treeString(treestr) //convert tree struct to string
      add tree to bundle
    } //end for
    attach number of trees to end of bundle
    MPI_Send( bundlestr, 0, EM_BUNDLE_OPT)
    resetBestTree()                 //clear bestlist
    MPI_Probe(tag)                  //get tag of incoming msg
    MPI_Recv( treestr, 0, tag)   //recv starting tree for next round
    if(tag == EM_BEST_OPT) {
      str_treeReadLen(treestr)   //convert string to tree struct
    }
  } while(tag == EM_BEST_OPT)  //continue until tag=EM_STOP_OPT
} //end em_optimize


em_sendTreeforEval() {
  treeString(treestr)     //convert tree struct to string
  MPI_Send(treestr, nextSlave, TREE_TO_EVAL)  //send tree
  nextSlave += numExtraMasters  //determine next assigned worker
  if(nextSlave > (numProc-1)) {    //all assigned workers sent trees
    nextSlave = slaveRankStart + my_rank – 1  //send to first assigned worker
  }
} //end em_sendTreeforEval
```

Figure 5.10: Pseudo-code used by the extra masters (0 < rank <= # extra masters) when the multiple masters option is in effect.

```
waitForWork() {
  MPI_Probe(source, tag)        //get source rank, tag of incoming msg
  while (tag == TREE_TO_EVAL) { //msg is tree to eval
    MPI_Recv(treestr, source, TREE_TO_EVAL) //get tree string from assigned master
    str_treeReadLen(treestr)   //convert string to tree struct
    slaveEvalTree(source)      //eval and return to assigned extra master
    MPI_Probe(source, tag)   //get tag of next incoming msg
  } //end while
  //exiting while() assumes tag = QUIT_SLAVE
} // end waitForWork
```

```
slaveEvalTree(source) {
  smoothTree()         //branch length optimization
  evaluate()           //calc likelihood of tree
  treeString(treestr)  //convert tree struct to string
  MPI_Send(treestr, source, TREE_RESULT)  //send result to assigned extra master
} //end slaveEvalTree
```

Figure 5.11: Pseudo-code used by the workers (rank > # extra masters) when the multiple masters option is in effect.

The role of the worker process does not change with the addition of multiple masters. A minor modification was made to ensure that tree results are sent back to the appropriate extra master, rather than to the head master. To accomplish this, a source rank argument was added to the slaveEvalTree() routine, where results are returned following evaluation. This addition is evident from Figure 5.11.

From hm_makeDenovoTree() in Figure 5.9 it can be seen that the head master is responsible for timing and results output. Both the head master and extra masters loop through the while() loop in sync, each on the same round at any given point. Synchronization occurs as the head master waits to receive the results of a round from all extra masters before sending out the resulting best tree. Extra masters must wait to receive this starting tree before progressing to the next round. This same flow is also used for rounds of the do/while() loop in optimize(). Because the number of passes through while() in makeDenovoTree() is determined by the total number of taxa, the head master and extra masters exit together. This is not the case for the do/while() loop in optimize(), however, as the break from the loop is dependent on the overall results from the round, to which the extra masters do not have access. Thus, when the likelihood has ceased improving as determined by the head master, each extra master is sent a message containing the

EM_STOP_OPT tag when expecting the starting tree, indicating the end of the call to optimize(). Note that new routines hm_optimize() and em_optimize() were added for the same reasons as the new makeDenovoTree() routines.

The division of the master's responsibilities over the extra masters is not evident from em_makeDenovoTree(), but rather on the call to addTraverse(). This routine performs a traverse, in a specific order, over the nodes of the starting tree, calling testInsert() to generate a new topology at each point in the traverse. addTraverse() was modified slightly for the addition of multiple masters, as shown in Figure 5.10, though the same order of traversal was preserved for all extra masters. The 'treeCounter' variable is used to divide up the calls to testInsert(), where each new topology is generated and sent out. Each extra master uses its own 'treeCounter,' which increments by one with each point reached in the traversal, starting with zero on each new round. In order to evenly divide the workload of new topology generation over the extra masters, initially on the start of a new round each extra master only calls testInsert() when 'treeCounter' is equal to one less than that extra master's rank. testInsert() is not called again until 'treeCounter' has increased by an amount equal to the number of existing extra masters. The 'treeCounter' values for the first four calls to testInsert() for each of three extra masters are provided in Figure 5.8. The value of 'treeCounter' corresponds to the same topology for each extra master, thus allowing different topologies to be generated by different extra masters.

The workers are divided evenly over the extra masters in a similar fashion. This is evident from em_sendTreeforEval(), a new variant of sendTreeforEval() specifically designed for use by the extra masters. On each call to em_sendTreeforEval() from testInsert(), the rank

of the target worker is determined by increasing the rank of the previous target worker by the number of existing extra masters, where the initial worker's rank is equal to the sum of the number of existing extra masters and the rank of that extra master. This method of dividing up topology generation and workers over the extra masters is also utilized in em_optimize(), where addTraverse() is called from rearrange().

Once an extra master has generated and sent out all of its assigned topologies for a given round, it calls recvTrees() to receive the corresponding number of tree results. As it receives each evaluated tree, the extra master saves it to its 'bestlist' if worthy, otherwise discarding it. Once all results have arrived, those saved in 'bestlist' are bundled and sent to the head master. The same bundling process is used by the workers when keeping their best trees, with one difference. Here, the total number of trees generated by the extra master is placed at the head of the original bundle string format. The head master sums these values received from all extra masters in order to report the total number of trees evaluated in each round, which is part of program output.

5.3.2 Automatic Selection of Multiple Masters

As mentioned earlier, functionality was included in the program to automatically select the appropriate number of extra masters to maximize performance, which may be zero. Automatic selection is performed by default and may be turned off by the user. The user then has the option of entering a desired number of extra masters, or not engaging the multiple masters option. Allowing the program to determine the optimal number of extra masters is useful since multiple masters can be harmful to performance when an inappropriate number of extra masters is chosen. The user may not know how to select the

best number, and with the default setup the consequences of a poor decision are avoided. The number of extra masters is set automatically at program startup, and remains constant throughout program execution. Based on the results presented later in Chapter 7, it appeared that the most significant enhancement in performance occurred when roughly a quarter of the total selected processors were extra masters. It appeared that the number of taxa analyzed did not exert a significant influence on the associated performance under the conditions tested. Thus, this was not factored into the determination of the number of extra masters, nor was the number of sequence sites or extent of rearrangements. The goal here was to include functionality for automatic selection rather than have it accurately chose the best number of extra masters for the given conditions. The basis for automatic selection may be refined once results from larger systems and problems are available.

5.3.3 Performance Considerations

The addition of multiple masters has the potential for an enhancement or reduction in performance, depending on the situation. The most significant gain in performance results from parallelization of the master's responsibilities. These are ordinarily executed sequentially, even with a large number of processors available, causing a bottleneck to arise with large systems and problems. By selecting a larger number of extra masters when more processors become available, the improvement in performance is expected to scale with the size of system. The use of multiple masters is also expected to benefit performance in a manner similar to the option where workers keep best trees. In both cases, tree results are filtered in parallel, with low-scoring trees discarded, before reaching the master.

Appropriate selection of the number of extra masters is extremely important when using this option, as too large a number will lead to performance degradation. The most significant cause is most likely the loss of worker processes, whose roles are replaced by those of the extra masters. As a result, fewer processors are available for parallelization of the most computationally intensive steps of branch length and likelihood calculation. The presence of multiple masters also leads to an increase in the overall amount of work required. For instance, tree results must be received and analyzed twice, by both the extra master and head master. Extra steps are also added at the head master, such as the need to send each extra master the starting tree on every round, as well as the stop signal for each call to optimize(). The resulting performance degradation is likely to be more noticeable with fewer trees, as the benefits of parallelization are not sufficient to offset the extra efforts.

Each of the three optimizations presented in this chapter has been described individually, under the assumption that the other two optimizations were not in effect. The intention, however, is to use all three optimizations together, further enhancing performance over their individual contributions. The optimizations were implemented as options easily turned on and off by the user, in any combination. Details on the selection of each option are provided in the next chapter, 'The Software.'

Chapter 6 The Software

The pseudo-code presented in Chapters 4 and 5 was incorporated into the existing fastDNAml code. Additional user options were added to allow user selection of the three optimizations. This chapter provides a brief description of the associated input and output formats, focusing on the additions made for the optimizations.

6.1 Input Format

The input format used for the program developed for this thesis was preserved from fastDNAml, and the reader is referred to the associated user documentation [24]. A similar input format is used with the PHYLIP package [10]. The user selects the optimizations introduced in this work following the same basic format for selecting the other user options available with fastDNAml. The available user options are those presented in Table 3.1. Although the input format is the same for this work, the modification was made to read the input from a file rather than standard input. A sample input file is provided below in Figure 6.1.

```
5  114 K G X M P
K 2
M 2
P 5
Sequence1   CACGGTGTCGTATCATGCTGCAGGATGCTAGACTGCGTCANATGTTCGTACTAACTGTG
Sequence2   CGCGGTGTCGTGTCATGCTACATTATGCTAGACTGCGTCGGATGCTCGTATTGACTGCG
Sequence3   CGCGGTGCCGTGTNATGCTGCATTATGCTCGACTGCGRCGGATGCTAGTATTGACTGCG
Sequence4   CGCGCTGCCGTGTCATCCTACACGATGCYAGACAGCGTCAGCTGCTAGTACTGGCTGAG
Sequence5   CGCGCTGTCGTGTCATACTGCAGGATGCTAGACTGCGTCAGCTGCTAGTACTGGCTGAG

            AGCTCGATGATCGGTGACGTAGACTCAGGGGCCATGCCGCGAGTTTGCGATGCG
            AGCACGGTGATCAATGACGTAGNCTCAGGRTCCACGCCGTGACTTTGTGATNCG
            AGCACGATGACCGATGACGTAGACTGAGGGTCCGTGCCGCGACTTTGTGATGCG
            ACCTCGGTGATTGATGACGTAGACTGCGGGTCCATGCCGCGATTTTGCGRTGCG
            ACCTCGATGCTCGATGACGTAGACTGCGGGTCCATGCCGTGATTTTGCGATGCG
```

Figure 6.1: Sample input file displaying user's selection of a 'bestlist' size of 2, global rearrangements, message packing with 5 trees per message, 2 extra masters, and workers keep their best trees.

The number of taxa for which sequences are provided is given as the first number in the file, followed by the number of sites in each sequence. On the same line, characters are entered representing the user's selections of various options. Shown here are the 'K' and 'G' options preserved from fastDNAml, which select a nondefault size for the 'bestlist' and global rearrangements. The auxiliary data line, shown as the second line, is required to select a 'bestlist' size of 2. Following the user selections of 'K' and 'G' are selections for the three optimizations unique to this work. The 'X' option indicates that workers should keep their best trees, and no auxiliary data line is used. The size of the 'bestlist' is determined from the size entered with the 'K' option, which is 1 by default. The 'M' option indicates that the user wants to specify a particular number of extra masters, rather than use the automatic selection mechanism, which is the default in the absence of the 'M' option. When the 'M' option is used, an auxiliary data line is required to specify the number of extra masters. Here, 2 are selected. If the user wishes not to engage multiple masters, a '0' would be entered. Because the design of the multiple masters option requires at least one

worker for every extra master, if the user selects a number of extra masters inappropriate for the number of processors selected, the program prints a corresponding error message and terminates. The 'P' user option engages message packing, where the number of trees per message must be specified with an auxiliary data line. Here, a bundle size of five is selected. The program is designed to handle any combination of optimizations, in any input order, including the absence of any optimizations.

The taxa names and associated sequences are entered following any auxiliary data lines. A limited number of characters are allowed for the name, after which the sequence is entered, on the same line. Additional sequences are entered on subsequent lines. Note here that the interleaved format is used. It was found that when the input file is created with a program other than Notepad, such as MS Word or WordPad, the additional formatting used by these programs causes an error when the input file is run. The output produced from the input file in Figure 6.1 is provided in Figure 6.2.

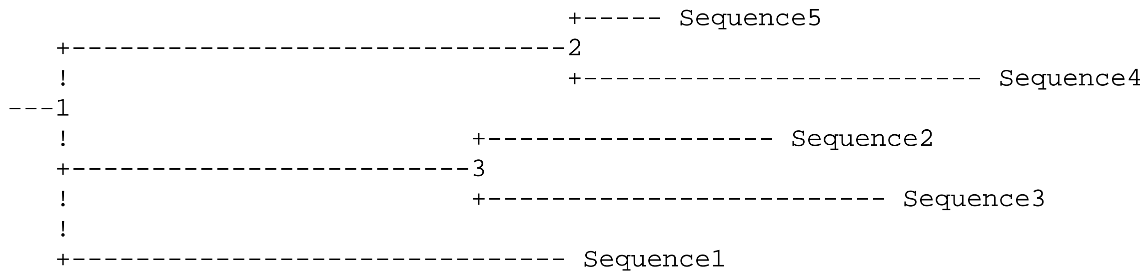
6.2 Output Format

Output for the program developed is printed to standard output. From the output provided in Figure 6.2, it can be seen that for each of the three optimizations selected, a notification is printed stating the selection, and any associated values such as the number of extra masters or bundle size. The total number of processors selected is also given. Execution times are printed at the very end of the output. Aside from these two additions, the output format for the program is identical to that of fastDNAml. The initial header was not changed from the original fastDNAml. A progress report indicating the insertion of each species is given, along with the associated number of topologies evaluated on each round.

The topologies, likelihoods, and branch lengths are printed for each of the trees in the final 'bestlist'. At the end of the output, a comparison of trees in the 'bestlist' is printed.

```
fastDNAml, version 1.2.2, January 3, 2000,
Copyright (C) 1998, 1999, 2000 by Gary J. Olsen

Based in part on Joseph Felsenstein's

   Nucleic acid sequence Maximum Likelihood method, version 3.3


5 Species, 114 Sites

Rearrangements of partial trees may cross 1 branch.
Rearrangements of full tree may cross 2 branches.

Total weight of positions in analysis = 114
There are 41 distinct data patterns (columns)

Empirical Base Frequencies:

   A       0.18570
   C       0.24823
   G       0.31783
  T(U)     0.24823

Transition/transversion ratio =   2.000000

(Transition/transversion parameter =   1.571835)

10 total processors selected

Option for workers to keep best trees in effect

Message packing option in effect with 5 trees per bundle

User selected multiple masters:
1 Head Master
2 Extra Masters
7 Slaves

Adding species:
   Sequence1
   Sequence2
   Sequence3
   Sequence1
      Tested 3 alt trees (INSERT)
      Ln Likelihood =    -336.11996
   Sequence5
      Tested 5 alt trees (INSERT)
      Ln Likelihood =    -365.80850
      Doing global rearrangements
      Tested 12 alt trees (BS)
```

Examined 21 trees

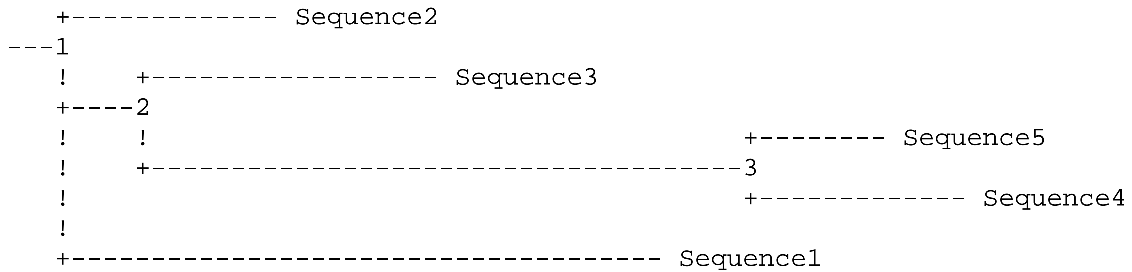


Remember: this is an unrooted tree!

Ln Likelihood =   -365.80850

Between	And	Length	Approx. Confidence Limits		
1	2	0.08775	(0.02504,	0.15802)	**
2	Sequence5	0.01873	(zero,	0.05038)	*
2	Sequence4	0.06742	(0.01738,	0.12217)	**
1	3	0.06940	(0.01393,	0.13070)	**
3	Sequence2	0.05346	(0.00521,	0.10607)	**
3	Sequence3	0.06177	(0.01061,	0.11786)	**
1	Sequence1	0.08289	(0.02281,	0.14988)	**

* = significantly positive, P < 0.05
** = significantly positive, P < 0.01



Remember: this is an unrooted tree!

Ln Likelihood =   -372.26531

Between	And	Length	Approx. Confidence Limits		
1	Sequence2	0.05076	(0.00335,	0.10237)	**
1	2	0.02085	(zero,	0.05426)	*
2	Sequence3	0.05492	(0.00578,	0.10858)	**
2	3	0.13281	(0.05866,	0.21777)	**
3	Sequence5	0.02831	(zero,	0.06896)	**
3	Sequence4	0.05946	(0.01021,	0.11325)	**
1	Sequence1	0.13541	(0.06223,	0.21911)	**

* = significantly positive, P < 0.05
** = significantly positive, P < 0.01

```
Tree also written to treefile.18652

Tree       Ln L         Diff Ln L       Its S.D.   Significantly worse?

  1    -365.80850  <------ best
  2    -372.26531      -6.45680        6.6552            No


TOTAL EXECUTION TIME FOR INSERTION PHASE = 0.200366 seconds
TOTAL EXECUTION TIME FOR GLOBAL REARRANGEMENTS = 0.165016 seconds
```

Figure 6.2: Output produced from the input file provided in Figure 6.1.

6.3 Program Execution

The program developed for this work was given the name fastDNAml_jbg, with the C source code provided in the file fastDNAml_jbg.c with the associated header file fastDNAml_jbg.h. Since MPI was used to implement the parallelization, the commands 'mpicc' and 'mpirun' are used for compiling and executing the program, respectively. For the RIT Cluster with input file 'input.phy', the commands used would be:

mpicc fastDNAml_jbg.c –o fastDNAml_jbg –lm

mpirun –machinefile nodelist –nolocal –np 10 fastDNAml_jbg input.phy

Note here that 10 total processors are selected. The 'nodelist' represents a list from which the desired processors are selected. For this thesis, all 94 processors were included in the list. The use of '-nolocal' specifies that the head nodes, from which the program is launched, should not be used to execute the program. It should be noted that on

compilation, the warning ‘conflicting types for built-in function ‘malloc’’ is printed. This is an artifact from the original fastDNAml program and should be ignored.

The user options associated with the new optimizations were engaged in various combinations in order to generate the results presented in Chapter 7. The execution times for the two phases printed to the output provided the results demonstrating the effects of the optimizations on performance. The topology results, along with the likelihood and branch lengths, were not reported in the results, as the focus was on the execution times. It is noted, however, that the program developed produced results identical to those of fastDNAml.

Chapter 7 Results & Analysis

Once the optimizations at the core of this thesis work were incorporated into the parallelization of fastDNAml, the resulting executable program described in the previous chapter enabled generation of results used to judge the effectiveness of each optimization. In order to demonstrate the expected relief of the communication bottleneck, execution times and speedup for each optimization are presented individually, as well as in combination, alongside results from the unoptimized parallelization. An analysis of the results is given, including potential reasons for any unexpected results. An investigation of how well the results compare to those of fastDNAml is also provided.

7.1 Experimental Conditions

Since this work focuses on the effects of the optimizations to parallelization, other factors known to influence results, both in terms of topology and execution time, were held constant. A single dataset containing 50 taxa, each with 773 sites, was used for all results presented in this chapter. The dataset is a subset of a dataset of nuclear ribosomal plant DNA published by Hu et al. [16], available through the online TreeBASE database [43]. Although the optimizations are targeted for the analysis of large-scale problems, with datasets containing well over 50 taxa, generating results for such a dataset would prove to be too time-consuming. Therefore, a smaller dataset is used for demonstration. The default program options were preserved from fastDNAml, including site-specific weights, rates of evolution, etc., with only the size of the 'bestlist' and extent of rearrangements explicitly selected. A 'bestlist' size of 15 trees is used, along with local rearrangements crossing 1 branch and global rearrangements crossing 47 branches, where the value of 47 was

obtained through the typical method of subtracting three from the total number of taxa. Such rearrangements result in the generation of a maximum of 95 trees per round during the insertion phase and 8,742 trees per round on global rearrangements, with a dataset of 50 taxa.

Results were generated through execution on the RIT Research Computing Cluster [26], using 3, 5, 10, 20, 30, 40, 50, 60, 70, 80, and 94 total processors, where applicable. The sequential execution time necessary to determine the speedup associated with parallelization was obtained through execution of the program on one processor. The cluster was found to show slight variation in execution time on multiple runs under the same conditions. Most of the data points presented were therefore obtained from an average of two or three runs generated at different times of the day and week to account for a fluctuating cluster load. For each set of results, the execution times were obtained for both the insertion phase, involving taxon insertion and local rearrangements on partial trees, and the global rearrangements phase, where extensive branch swapping is performed on full trees. The exact code portions corresponding to the two phases is evident from the pseudo-code presented in previous chapters. The execution times of the two phases were summed to obtain the time for the entire tree-building process.

7.2 Parallelization without Optimization

Under the conditions given above, results were obtained for the unoptimized parallelization described in Chapter 4, and are provided below. The variation in execution times with differing numbers of processors can be seen from Figure 7.1. With parallelization, one would typically expect the execution time to continue to decrease with an increase in the

number of processors used. Here, however, the execution time decreases only very slightly as the number of processors increases from 30 to 94. The expected drop is seen initially, when fewer numbers of processors are utilized.

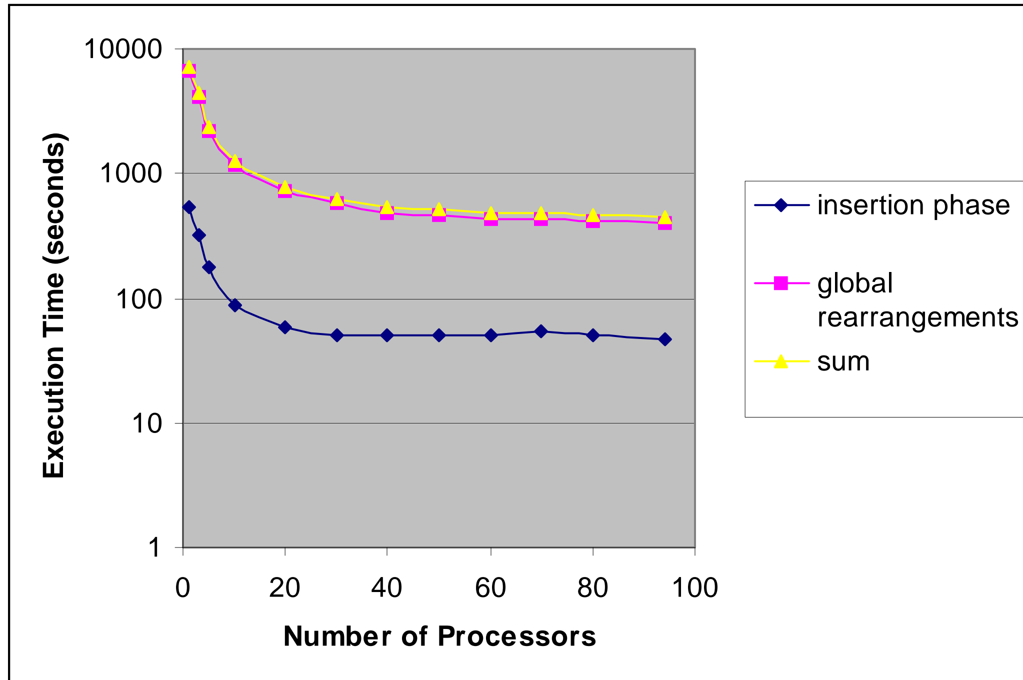


Figure 7.1: Execution time over a varying number of processors for the two phases of the tree-building process, and their combination, obtained for parallelization without optimization.

The factor by which performance improves relative to sequential execution for varying numbers of processors is shown by the speedup presented in Figure 7.2. The dashed line represents perfect scaling, which is the ideal case where execution time drops by a factor equal to number of processors used, relative to sequential execution. This ideal is rarely achieved due to several factors, including parallelization overheads. However, near-perfect

scaling is achievable, and is the typical goal. From the results in Figure 7.2, however, it is clear that the unoptimized parallelization falls far from this goal.

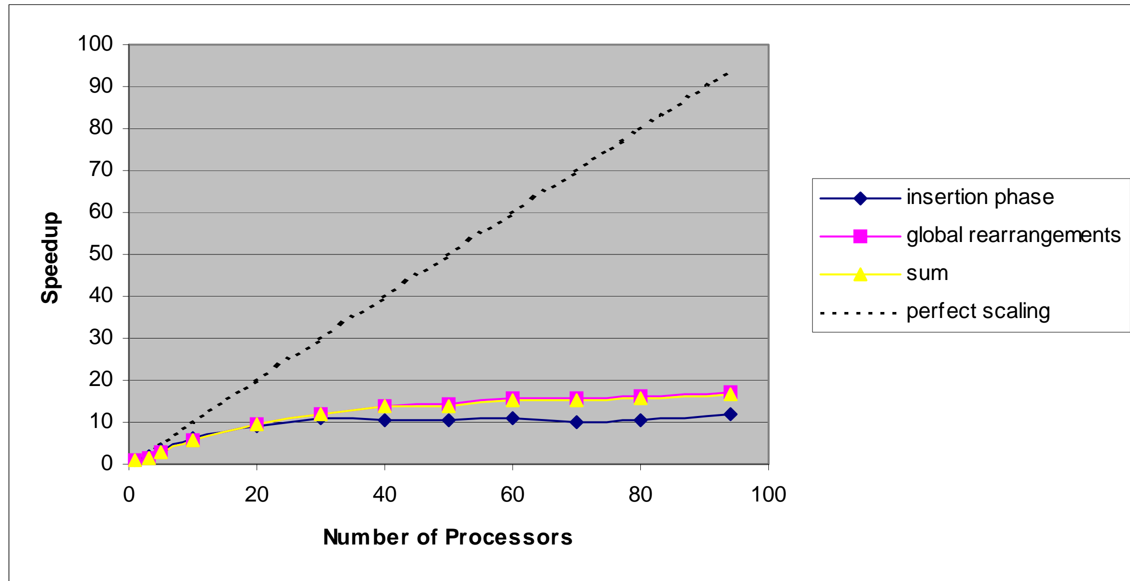


Figure 7.2: Speedup for the two phases of the tree-building process, and their combination, obtained for parallelization without optimization.

This trend is characteristic of the presence of a bottleneck, which limits the potential performance gained as the number of processors increases. It is believed that the bottleneck resulting from a single master arises at two main points during a given round of topology generation and evaluation. The bottleneck initially appears when newly generated topologies are sent out by the master to the workers for evaluation. If not all workers remain busy at all times, the performance one expects to gain from the availability of additional processors is reduced. Two major situations may lead to idle workers. First, when there are more workers than trees in a given round, some workers will go unused.

Secondly, the master may not generate new topologies at a rate sufficient to keep up with the demands of the workers. This would occur initially as workers wait for their first tree to evaluate in a round, as well as when a worker completes evaluation of a tree before receiving the next. With more processors, and thus workers, available, the more time is spent by each worker waiting for its next tree. The other point during a given round where a bottleneck is expected to arise is on receipt of all evaluated trees at the master. As the number of trees in a given round increases, the bottleneck is expected to worsen, since the master will require more time to handle the incoming results while workers sit idle.

These bottlenecks should be taken into consideration when explaining the results presented in Figures 7.1 and 7.2. The bottleneck occurring on topology generation, for instance, may explain the reduced speedup seen during the insertion phase as compared with global rearrangements. It is likely that during a significant portion of the insertion phase there are fewer trees generated per round than the number of available workers, since the number of potential topologies increases with the number of taxa inserted thus far. Global rearrangements, on the other hand, produce a sufficiently larger number of trees than workers, and the bottleneck is more likely to result from the master's inability to meet the demands of its workers rather than from unused workers. Also notice that since the execution time for global rearrangements is significantly longer than that for the insertion phase, the time for the entire tree-building process, indicated in the figures as the 'sum,' is dominated by global rearrangements.

With the conclusion that the results obtained for the unoptimized parallelization suffer partially from the bottleneck arising during topology generation, a quantification of the

relative time needed for the generation of a topology compared with its evaluation is warranted. The calculation of the branch lengths and likelihood for a given tree is known to be the most computationally intensive step of maximum likelihood-based phylogenetic inference. It was unknown, however, how the time needed for the generation of a topology compares with that for its evaluation. Thus, these two calculations were timed separately for a run of the unoptimized parallelization using 30 of the 50 taxa in the original dataset. The time needed for a call to insert() represents the time to generate a topology, whereas the combined time for the calls to smoothTree() and evaluate() represents the time taken for the branch length and likelihood calculation. The times for each of these sections were summed over all 7,340 trees generated on the run, and a resulting time of 5.3 seconds was achieved for all topology generations, compared with 448.3 seconds for all branch length and likelihood calculations. Thus, the time required to evaluate a tree is greater than that required for its generation by a factor of roughly 85. The evaluation of an individual tree serving as the parallelized entity thus remains justified, although topology generation coupled with other parallelization overheads could potentially lead to the inability of the master to keep up with the workers' demands as the number of available processors increases.

7.3 Message Packing

Results were obtained with the message packing option in effect for three different message sizes, specifically 10, 5, and 3 trees per message bundle. Roughly the same effects on execution time, and thus speedup, were seen for all three message sizes analyzed. Message packing was found to affect the execution time for the insertion phase, but not for global rearrangements. Unfortunately, from the results it appears that an unexpected negative

effect on performance results, leading to higher execution times compared to the unoptimized parallelization, for most numbers of processors tested. For small numbers of processors, mainly three and five, the execution time for the insertion phase remains approximately the same with or without message packing for all message sizes. With the smallest message bundle size of three trees, however, this equality is maintained through 20 processors. It appears that the negative effect on the execution time of the insertion phase is relatively constant as the number of processors increases from 30. The associated execution times and speedups for the three message sizes tested are displayed in the graphs in Figures 7.3-7.8 below.

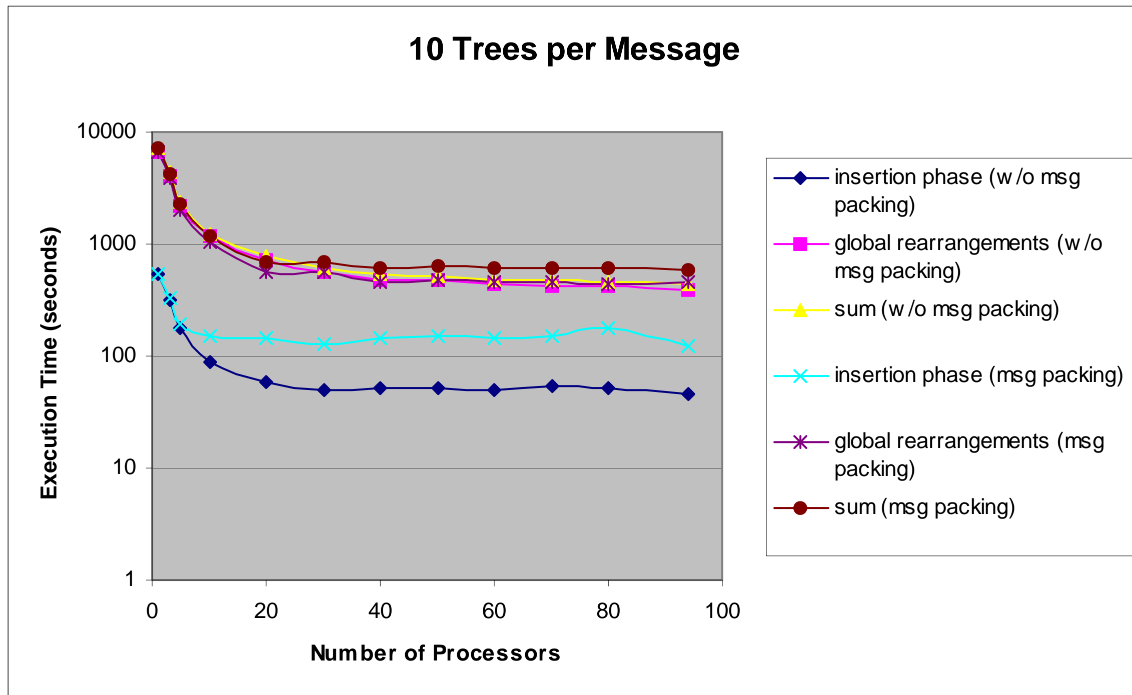


Figure 7.3: Execution times for message packing with 10 trees per message, compared with those without message packing from section 7.2.

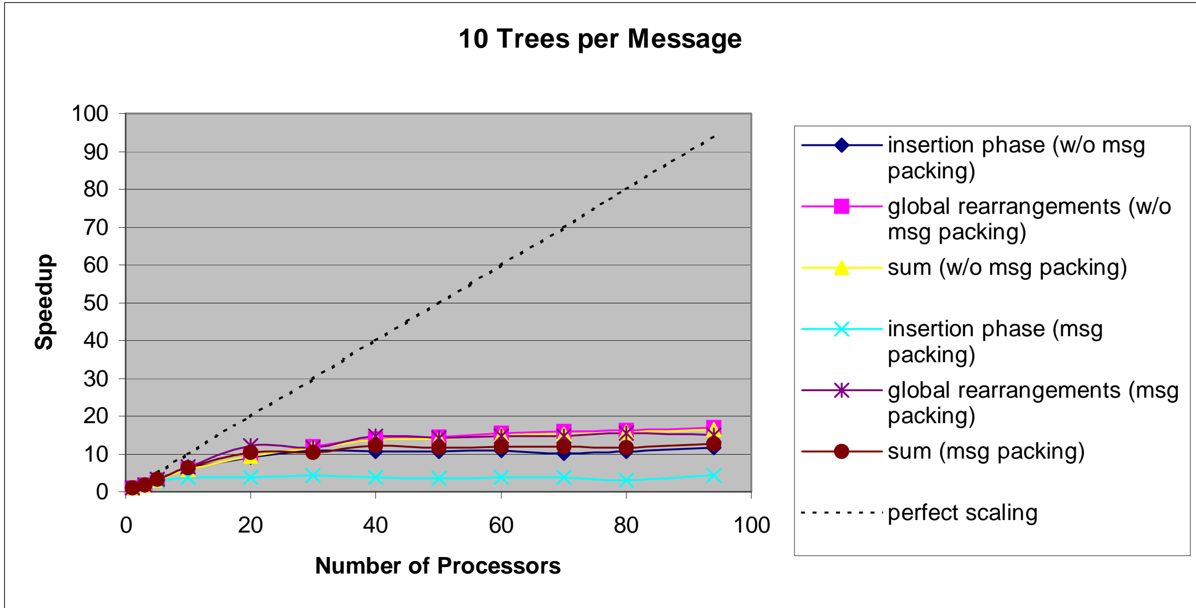


Figure 7.4: Speedup for message packing with 10 trees per message, compared with speedup in the absence of message packing from section 7.2.

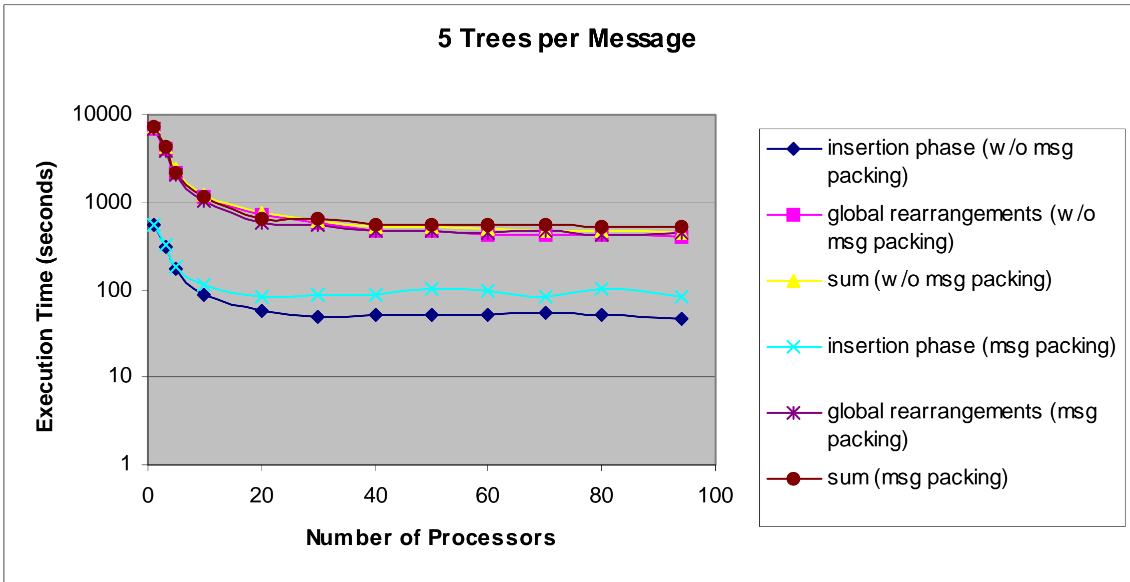


Figure 7.5: Execution times for message packing with 5 trees per message, compared with those without message packing from section 7.2.

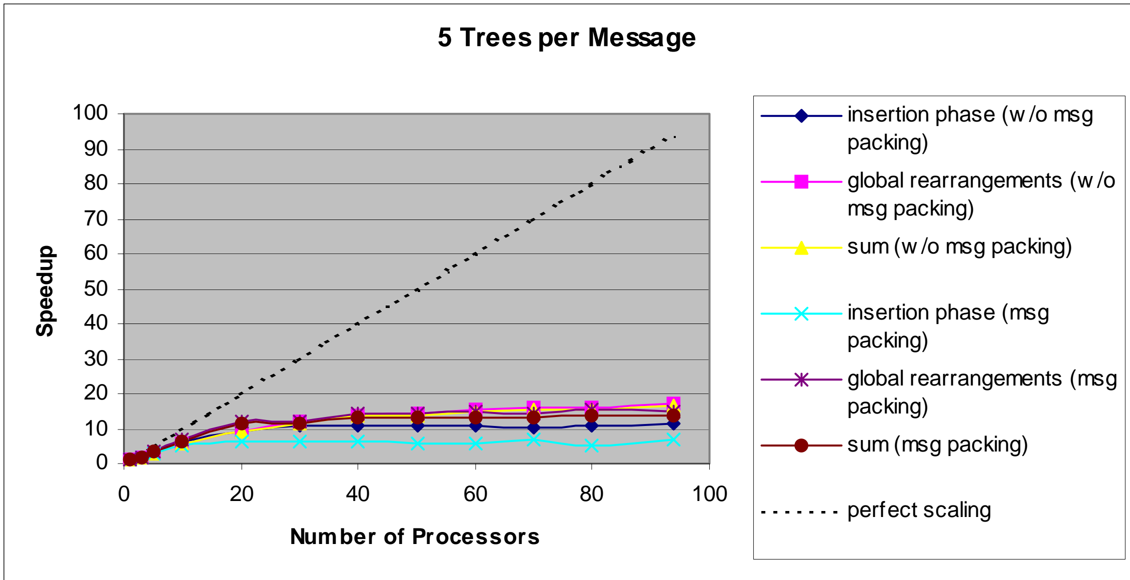


Figure 7.6: Speedup for message packing with 5 trees per message, compared with speedup in the absence of message packing from section 7.2.

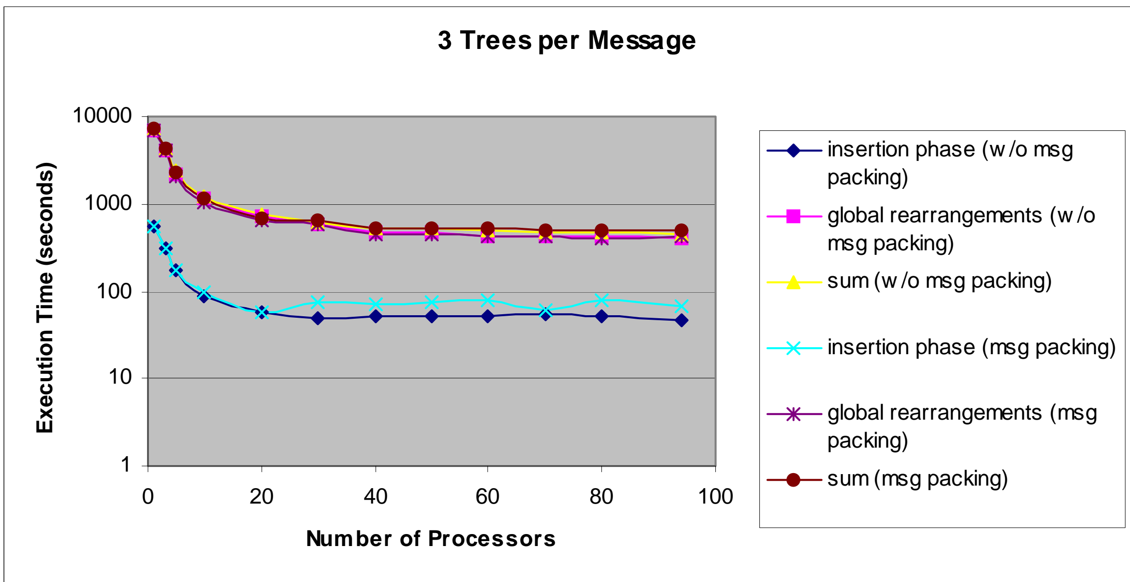


Figure 7.7: Execution times for message packing with 3 trees per message, compared with those without message packing from section 7.2.

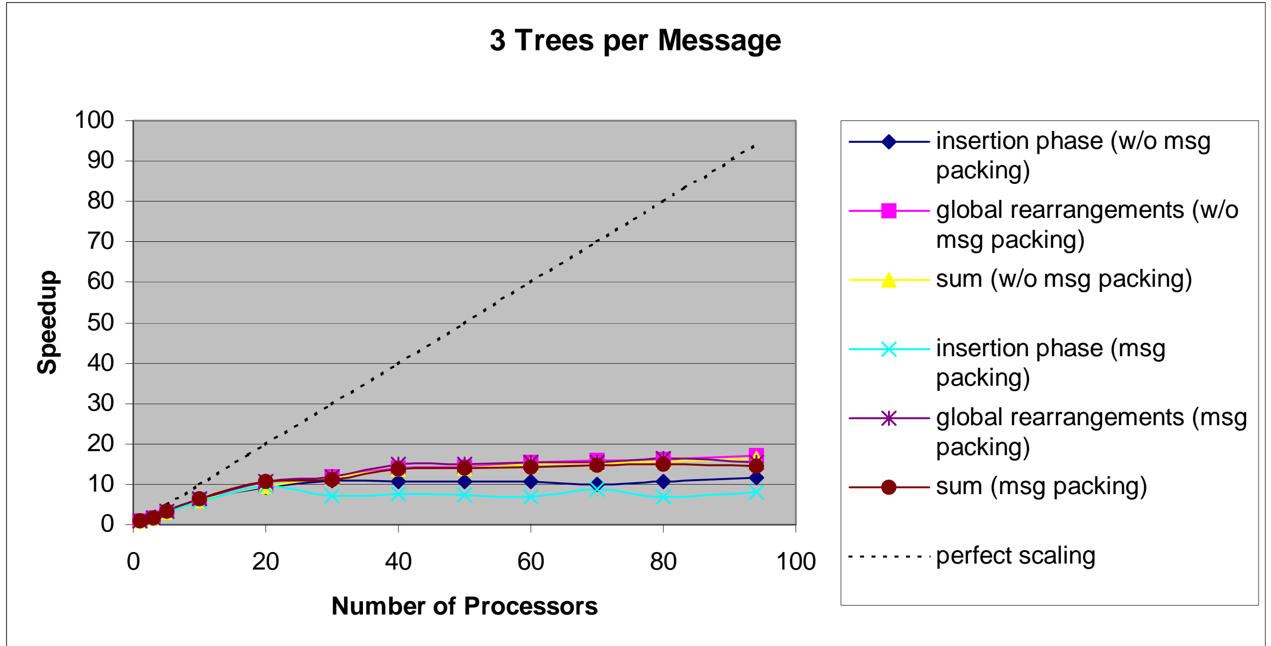


Figure 7.8: Speedup for message packing with 3 trees per message, compared with speedup in the absence of message packing from section 7.2.

The reduced performance seen with message packing can be explained when considering the previous conclusion that the master is unable to keep all workers busy at all times. As mentioned previously, when insufficient numbers of trees are produced on a given round, some workers go unused. Message packing exacerbates this condition, increasing the minimum number of trees required to occupy all workers by a factor equal to the bundle size. This explains the more noticeable effect on the insertion phase, where fewer trees are available for evaluation in a given round. Also likely to be a contributing factor is the increase in the amount of time before each worker receives its first bundle of trees to evaluate at the beginning of each round. The reduced overheads thought to result from message packing are likely not sufficient to offset these negative effects when the number of trees is small.

The lack of any noticeable impact on global rearrangements is noteworthy. During this phase, far more trees are produced per round than during the insertion phase. The results suggest that the number of trees produced per round on global rearrangements may lead to fewer overheads, offsetting the negative effects of message packing that were seen during the insertion phase, under the conditions analyzed. It is very possible that an inadequate number of trees were produced with the relatively small dataset used to generate these results. As the optimizations presented in this thesis were intended for large-scale problems, the proposed reduction in overheads due to message packing may appear under such conditions, leading to the expected performance enhancement.

Although varying the bundle size does not appear to have a pronounced effect on the results under the conditions tested, a discussion is still warranted. Figure 7.9 below provides a comparison of the execution times for the combined insertion phase and global rearrangements for the three bundle sizes under analysis. Note that the vertical scale used in this figure shows more detail than seen in the above figures. Interestingly, with this increased magnification it was discovered that for smaller numbers of processors, message packing actually appears to benefit performance, though the effect is minor. This further suggests that too few trees are generated with the selected dataset to fully utilize all available workers, and that message packing may potentially lead to significant performance gains when greater numbers of trees are analyzed with large-scale problems.

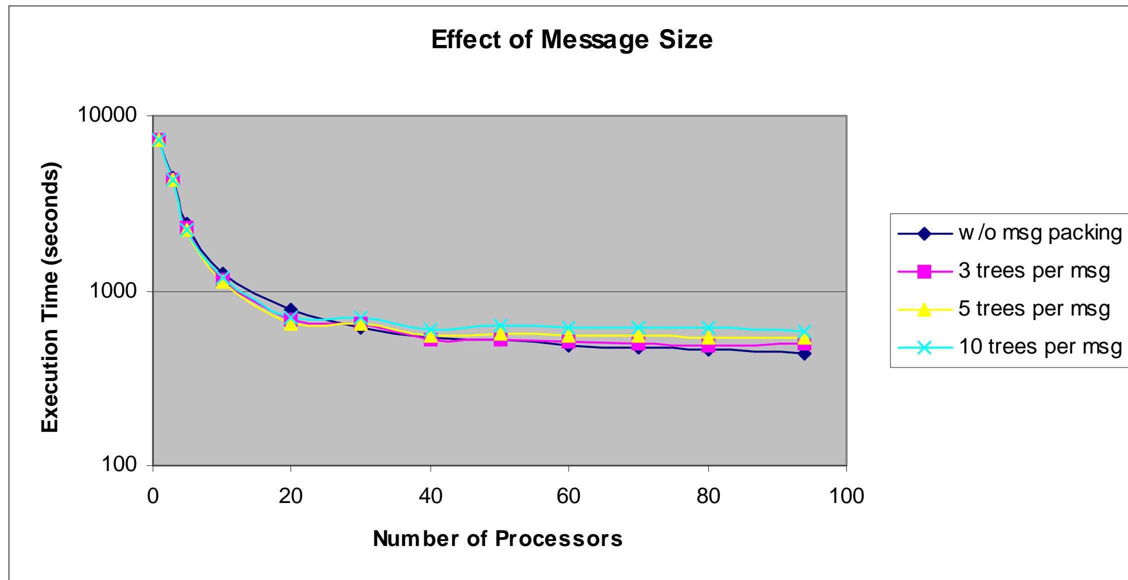


Figure 7.9: Execution times for combined insertion phase and global rearrangements for varying message sizes.

When comparing the execution times between the three bundle sizes in Figure 7.9, it appears that the differing sizes lead to slight variations in the times for both phases, with a larger difference among sizes seen in the insertion phase. To avoid a cluttered figure, only the values for the sum of the execution times are plotted. It is apparent that larger bundle sizes have a more pronounced negative effect than do smaller bundle sizes, and that the differences appear to increase with the number of processors. There is the possibility, however, that since a larger bundle size leads to a greater reduction in performance, that it may also have the potential to lead to a greater enhancement in performance, when sufficient numbers of trees are available. Thus, evaluating the performance of message packing with larger problem sizes is recommended.

7.4 Workers Keep Best Trees

Results obtained through the optimization where workers keep their best trees are provided below in Figures 7.10 and 7.11.

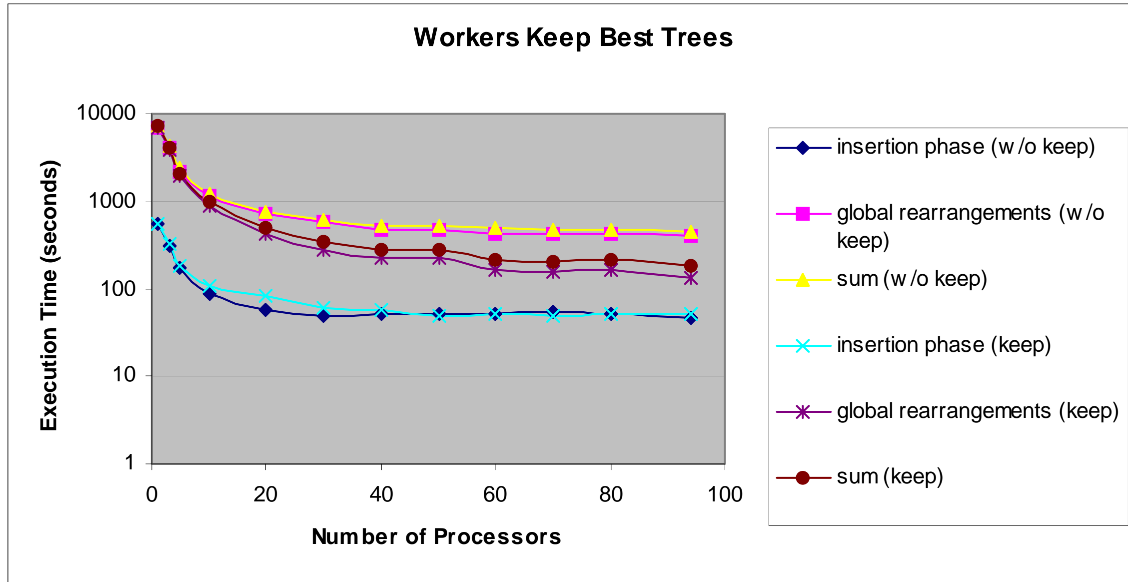


Figure 7.10: Execution times when workers keep their best trees, compared with the unoptimized parallelization from section 7.2.

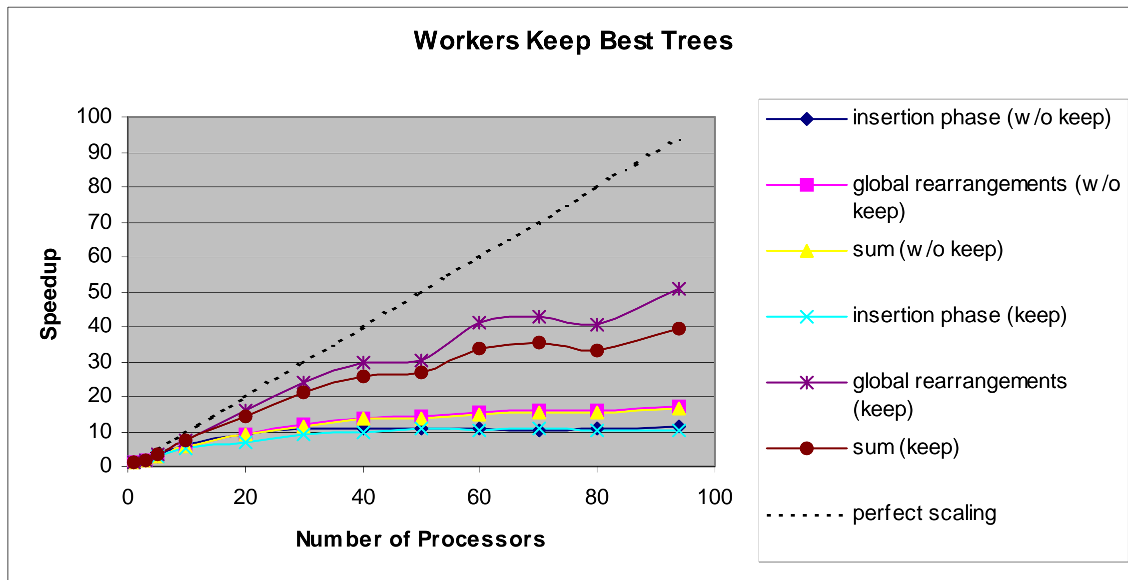


Figure 7.11: Speedup when workers keep their best trees, compared with the unoptimized parallelization from section 7.2.

From these results, it appears that no effect on the insertion phase occurs when workers keep their best trees. However, a dramatic improvement in performance is seen during global rearrangements. The factor by which the speedup improves over the unoptimized parallelization increases with the number of processors. However, the results still fall far short of perfect scaling. The results suggest that a bottleneck is arising when the single master handles all incoming tree results. With the bottleneck partially relieved through this optimization, the performance more closely resembles that expected for parallelization.

The absence of a performance gain during the insertion phase is expected, as significantly fewer trees are evaluated per round. It is thought that for much of insertion phase, there are insufficient numbers of trees produced on each round to fill up all workers' 'bestlist's, here given a size of 15. Thus, no trees are discarded and the master receives the same number of evaluated trees as in the absence of the optimization. If a small number of trees are discarded, it is likely that the overheads required for this optimization offset any performance gained. Although an increase in the number of workers would result in fewer trees discarded, the effects are not apparent from the results due to the consequent increase in the efficiency of parallelization.

No good explanation could be found for the presence of the dips in speedup with 50 and 80 processors. Initially, variations in the cluster were suspected, but this same pattern appeared in two sets of data generated several days apart. The combined effects on performance over varying numbers of processors from differing numbers of trees discarded and differing extents of parallelization may be at fault. Gathering results with varying sizes of the 'bestlist' may offer some insight. However, due to time constraints this was not possible.

7.5 Multiple Masters

Results were obtained for the multiple masters option using 3, 5, 8, and 10 extra masters. The graphs in Figures 7.12-7.15 below show the effects on execution time for these numbers of extra masters over a varying number of processors. It should be noted that in the graphs some of the data points for lower numbers of processors are absent, as enough workers to equal the number of extra masters must be available to run this option. For all four numbers of extra masters analyzed, the same general trend is seen in the execution times for global rearrangements, specifically longer execution times occur with lower numbers of processors, while shorter execution times are found with higher numbers of processors, as compared with parallelization using a single master. For the insertion phase, initially higher execution times eventually equalize with the case of a single master as the number of processors increases. However, the point where global rearrangements improve over a single master occurs at a lower number of processors than for the insertion phase. Of note concerning the insertion phase is that as additional extra masters are used, a greater impact is seen with less processors, but equalization is achieved earlier as the number of processors increases. Fewer extra masters show less of an impact, but the effect is seen with greater numbers of processors.

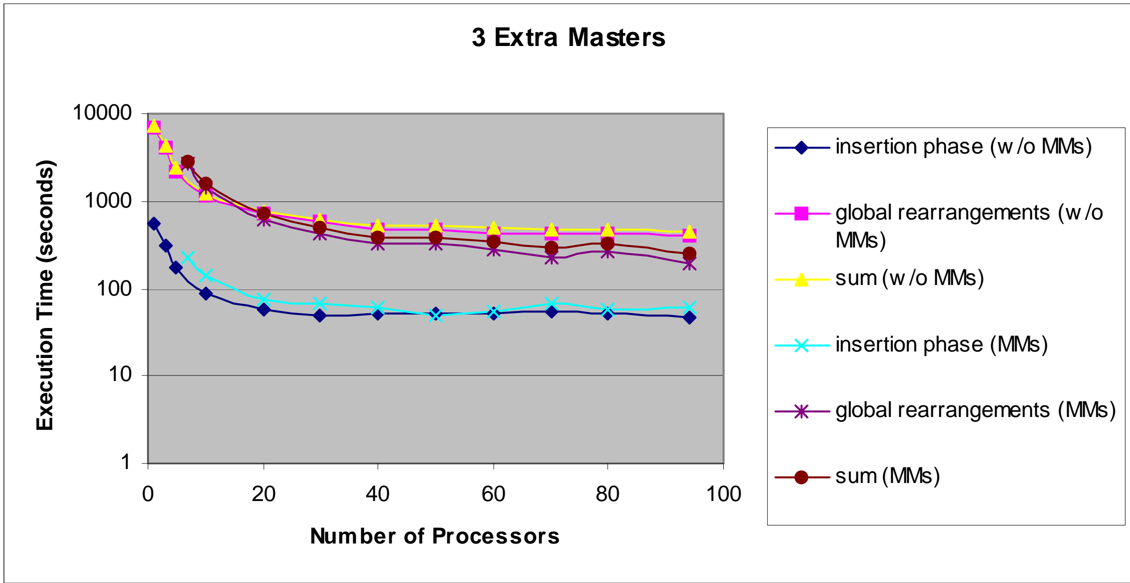


Figure 7.12: Execution times for 3 extra masters, compared with a single master from section 7.2.

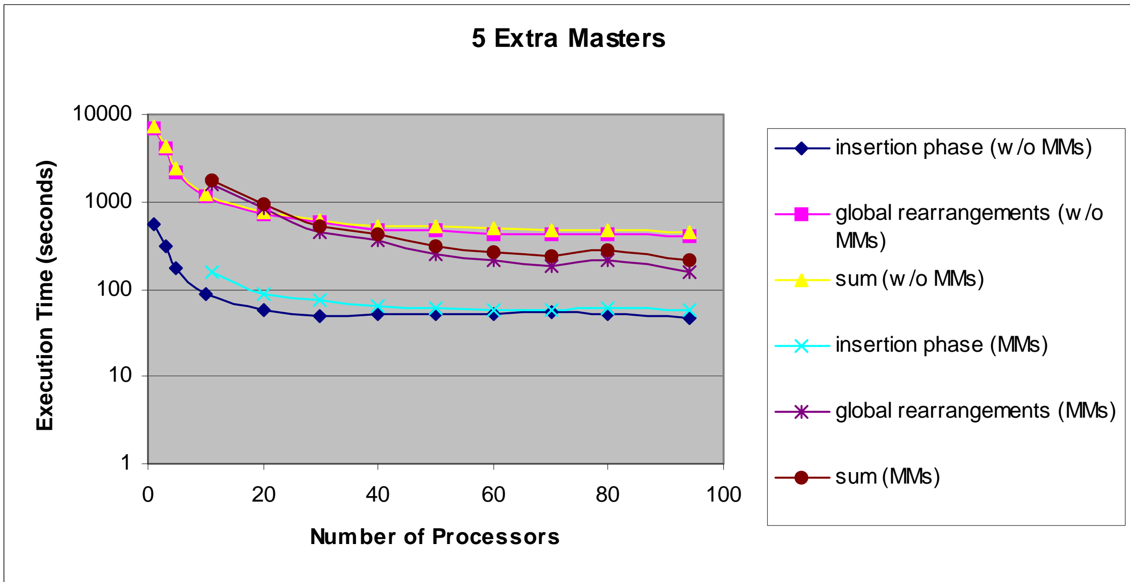


Figure 7.13: Execution times for 5 extra masters, compared with a single master from section 7.2.

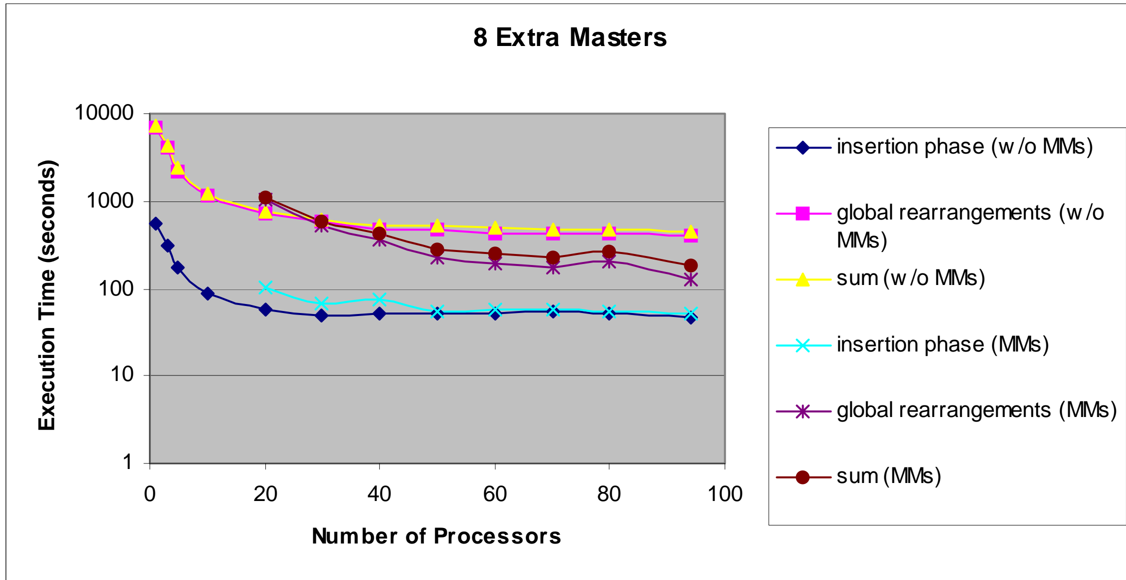


Figure 7.14: Execution times for 8 extra masters, compared with a single master from section 7.2.

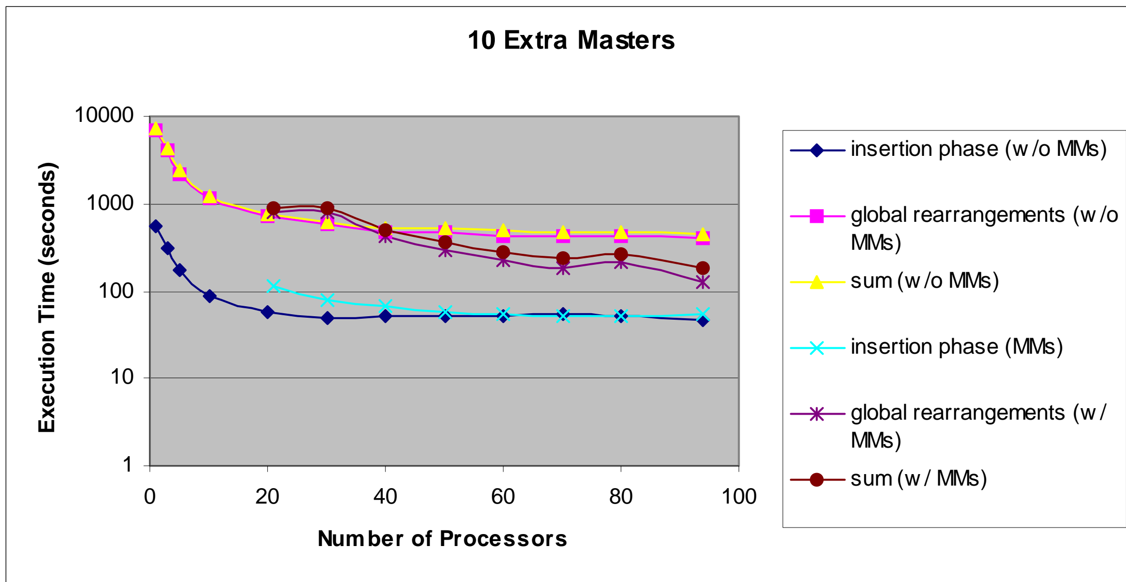


Figure 7.15: Execution times for 10 extra masters, compared with a single master from section 7.2.

With large numbers of processors, an increase in speedup was achieved for all numbers of extra masters used, with the factor by which speedup improved over a single master increasing with both the number of extra masters and the number of processors. It is

thought that an improvement in speedup is achieved through multiple masters due to relief of the bottlenecks arising both during topology generation and on the receipt of tree results. The reduction in performance with lower numbers of processors is expected, as less workers are available to parallelize the most computationally intensive step. A greater improvement on global rearrangements as compared with the insertion phase is also expected, as the larger number of trees produced during global rearrangements would otherwise lead to the bottleneck relieved through multiple masters. The associated speedup graphs for 3, 5, 8, and 10 extra masters are provided in Figures 7.16-7.19, respectively.

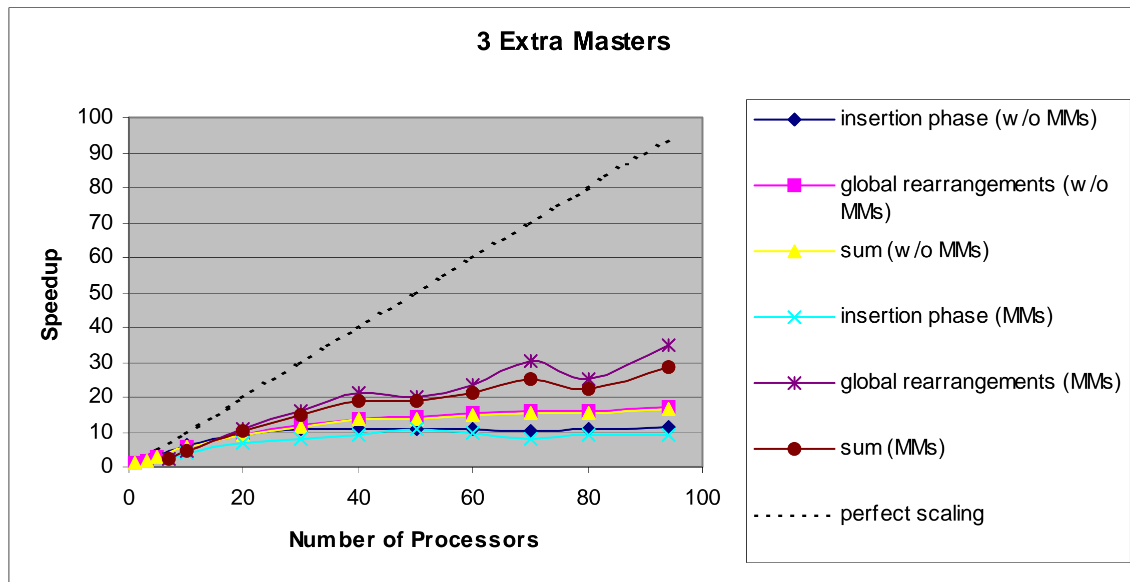


Figure 7.16: Speedup for 3 extra masters, compared with a single master from section 7.2.

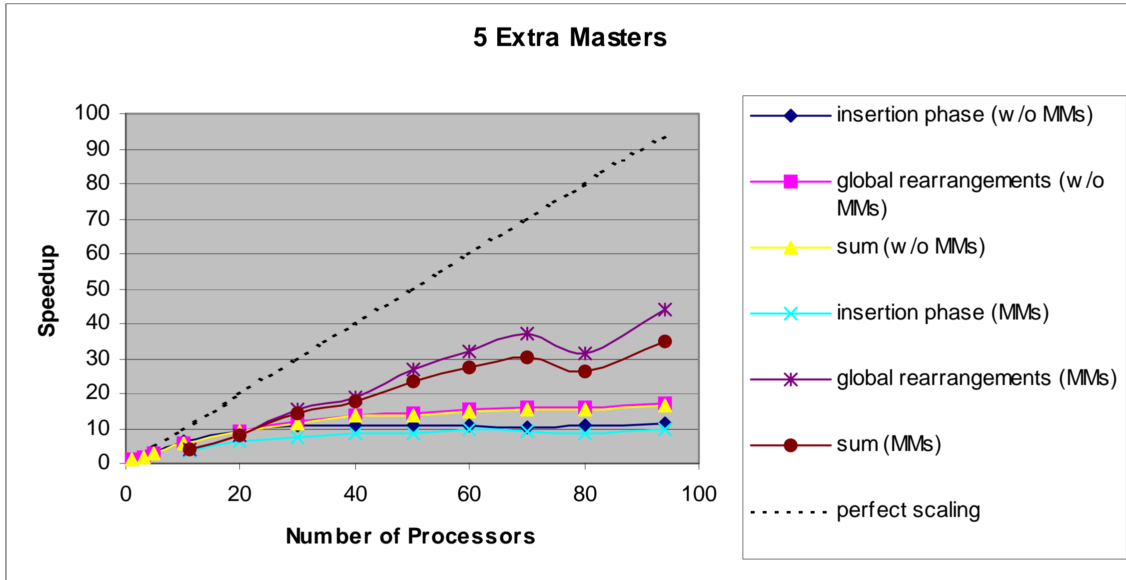


Figure 7.17: Speedup for 5 extra masters, compared with a single master from section 7.2.

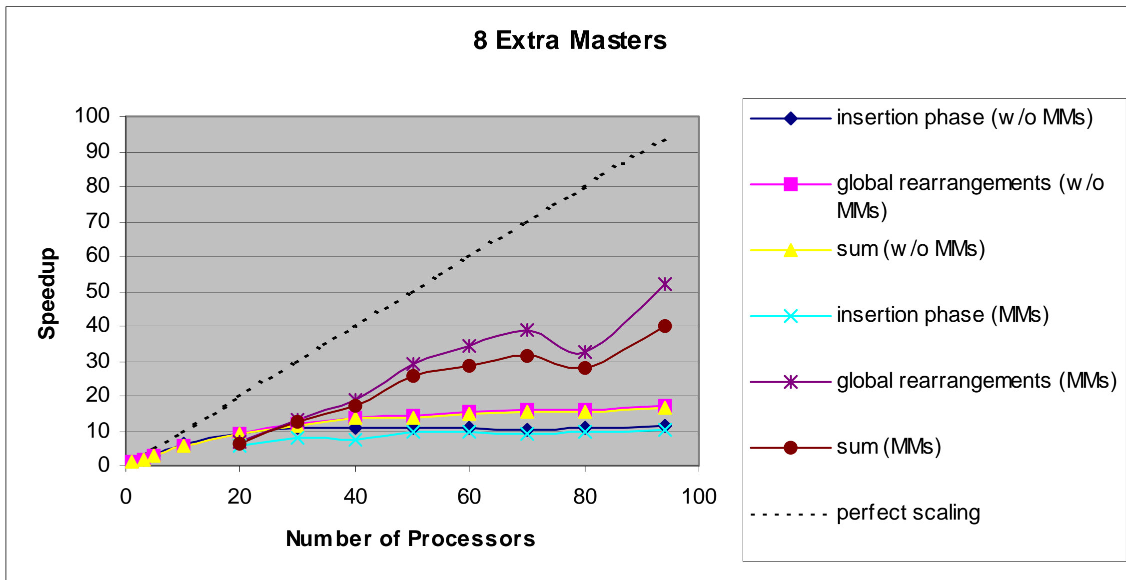


Figure 7.18: Speedup for 8 extra masters, compared with a single master from section 7.2.

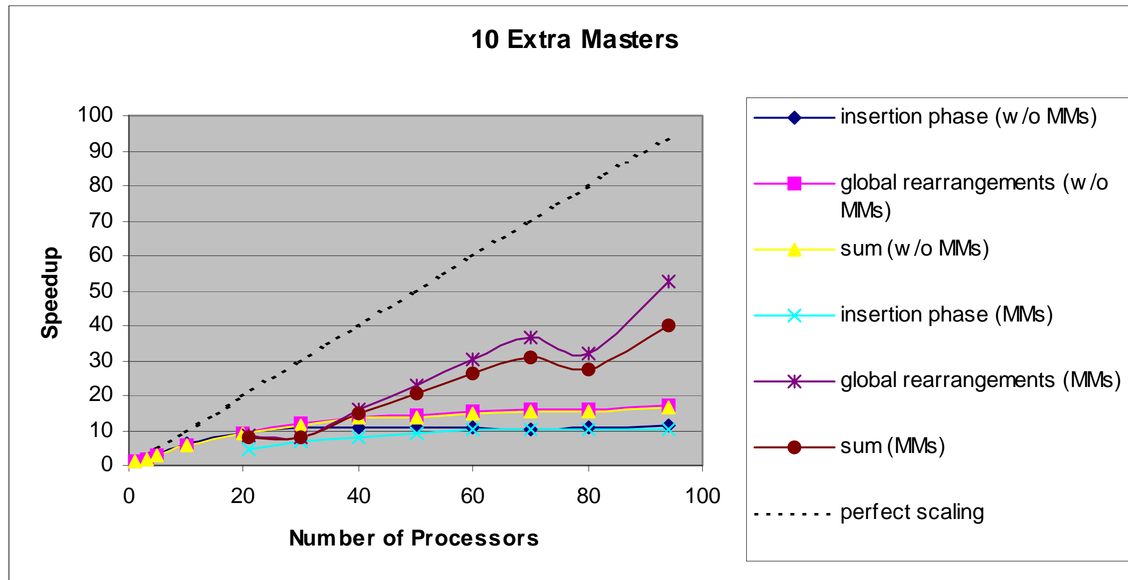


Figure 7.19: Speedup for 10 extra masters, compared with a single master from section 7.2.

A dip in speedup was seen for all four numbers of extra masters at 80 processors, similar to that seen when workers keep their best trees. Considering that the use of multiple masters causes evaluated tree results to be filtered in a way similar to that by workers when keeping their best trees, the dip may in fact be related to the combined effects of discarding trees and improved efficiency of parallelization, as previously described.

7.5.1 Automatic Selection of Multiple Masters

The effects on execution time of varying numbers of extra masters are compared over a varying number of processors in Figure 7.20 below. It can be seen that there is a distinct point where each number of extra masters becomes beneficial to performance over a single master. This point occurs with larger numbers of processors for higher numbers of extra masters.

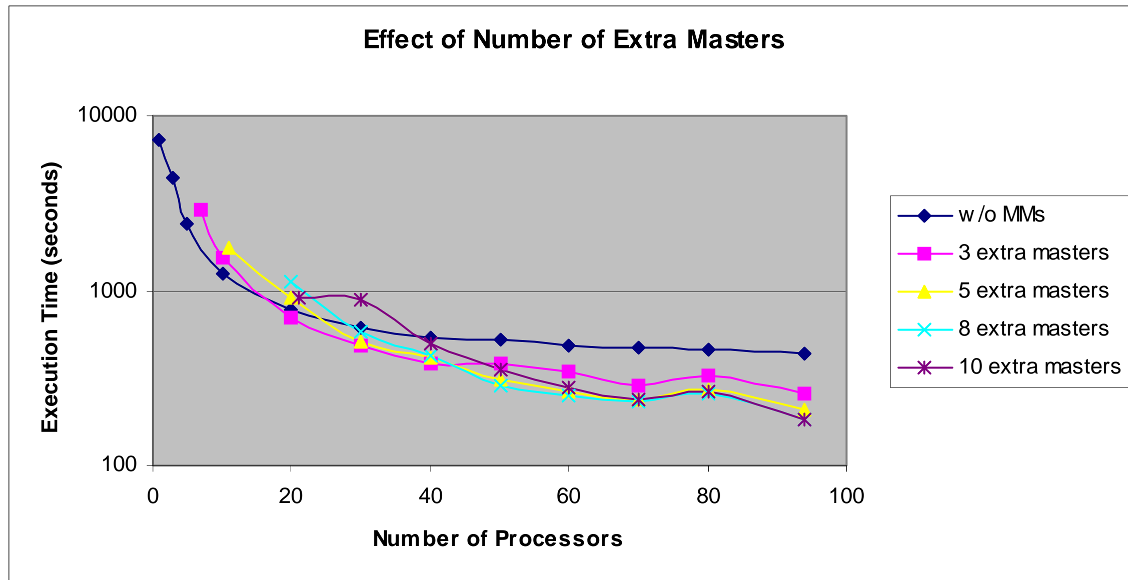


Figure 7.20: Execution times for combined insertion phase and global rearrangements for varying numbers of extra masters.

From Figure 7.20, it was estimated that the best performance results when a quarter of the total processors are extra masters. When 3 extra masters exist, for instance, performance is not enhanced until about 16 total processors are employed, as determined from Figure 7.20. Similarly, this value is 24 processors for 5 extra masters, 28 for 8 extra masters, and 38 for 10 extra masters, which roughly average to one extra master among every four processors. This rule is used to choose the number of extra masters when automatic selection of the number of extra masters is in effect, as described in Section 5.3.2.

Following incorporation of automatic selection for the number of extra masters, the results in Figures 7.21 and 7.22 were generated. The improvement in performance varies greatly with the number of processors. As a more linear improvement was expected, it is thought that the rule used for automatic selection does not maximize the potential gain in performance. The one in four rule was a very rough estimate, and may not be applicable for

all numbers of processors. Also, relatively small numbers of extra masters were used to determine the rule, considering that the one in four rule would suggest 23 extra masters be used with 94 total processors. Comparing Figures 7.22 and 7.19, however, it appears that the difference in performance gained from using the rule as opposed to a constant 10 extra masters is minimal for larger numbers of processors. Other factors are likely to influence the best number of extra masters for a given number of processors, and further investigation is warranted. However, the functionality for automatic selection was incorporated into the program, forming a foundation to which more refined selection mechanisms may be added.

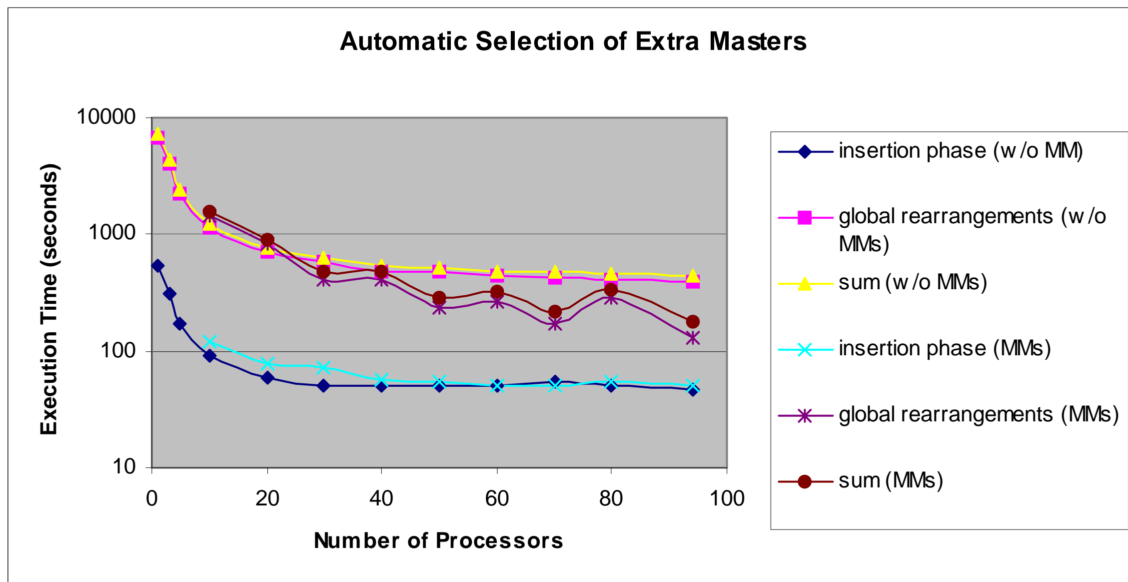


Figure 7.21: Execution times for automatic selection of extra masters, compared with a single master from section 7.2.

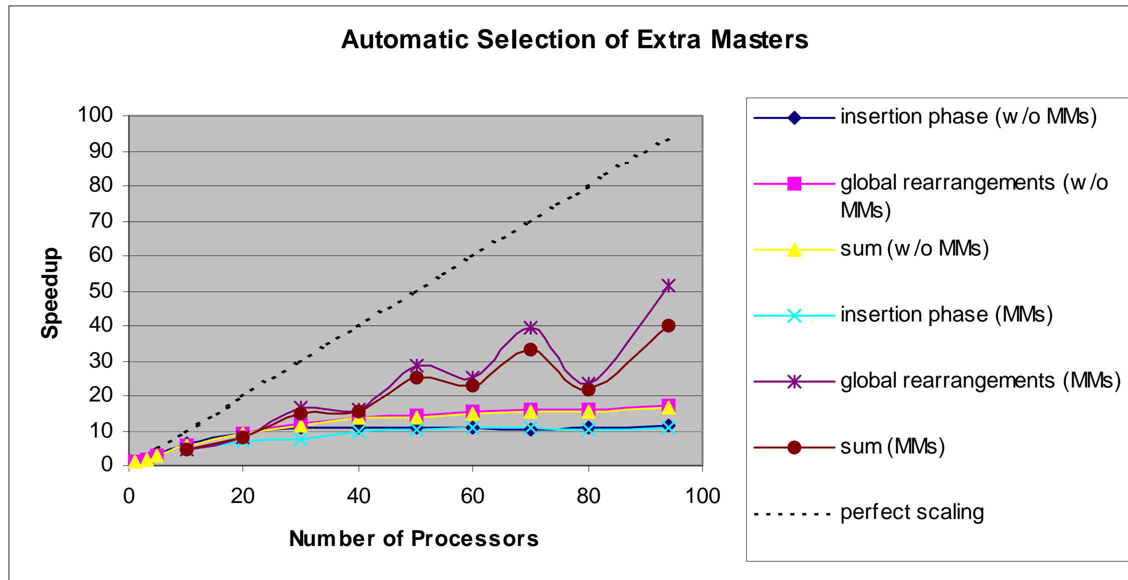


Figure 7.22: Speedup for automatic selection of extra masters, compared with a single master from section 7.2.

7.6 Combining & Comparing Optimizations

The optimizations incorporated into the parallelization of fastDNAml were designed to be used together, combining the enhancements in performance seen when used individually. Results obtained from combining optimizations are provided in Figures 7.23-7.26 below. The combination expected to lead to the most significant improvement in performance is that of multiple masters when workers keep their best trees. Automatic selection for the number of extra masters was employed. The associated results are provided in Figures 7.23 and 7.24, which show that the effects on performance are comparable to using multiple masters alone. Some of the performance gained when workers keep their best trees is lost on the addition of multiple masters. This result is disappointing, and suggests that perhaps the additional performance that would have been gained through the filtering of results by

the workers was already realized with the filtering performed at the extra masters. The deficiency of the automatic selection, however, may prevent demonstration of any benefits resulting from a combination of the two optimizations, as the most appropriate number of extra masters is likely not selected.

When all three optimizations were combined, using a bundle size of 5 for message packing, a slight decrease in performance was seen, which was not surprising as message packing alone lead to reduced performance. The results are provided in Figures 7.25 and 7.26.

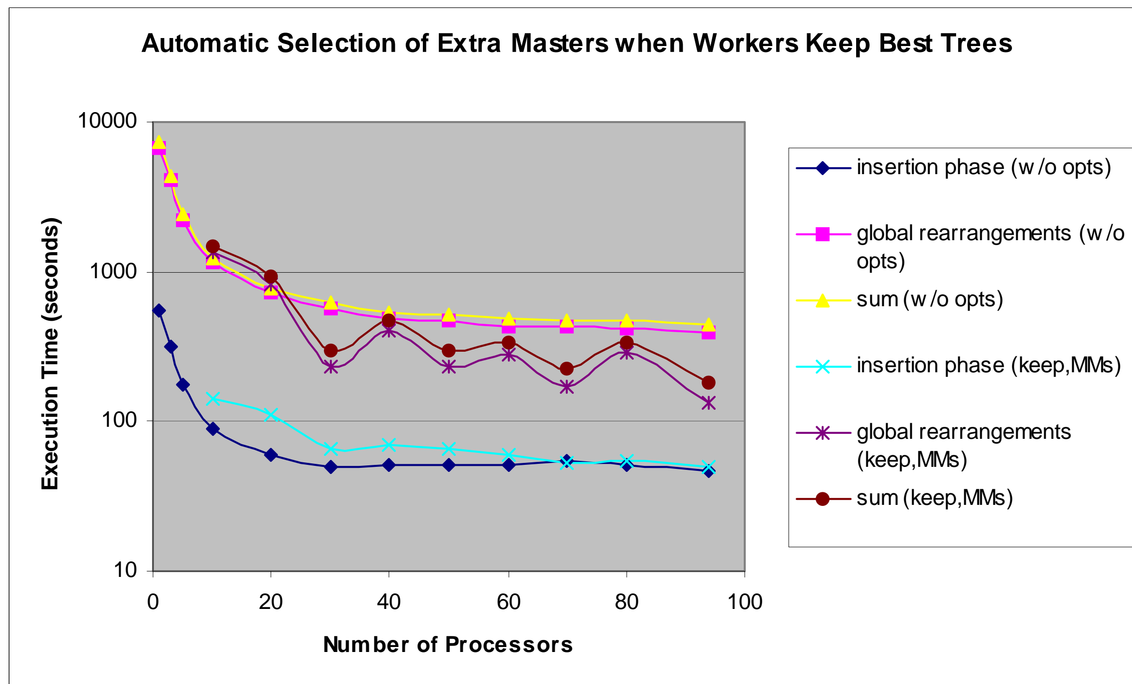


Figure 7.23: Execution times for automatic selection of extra masters combined with the option for workers to keep their best trees.

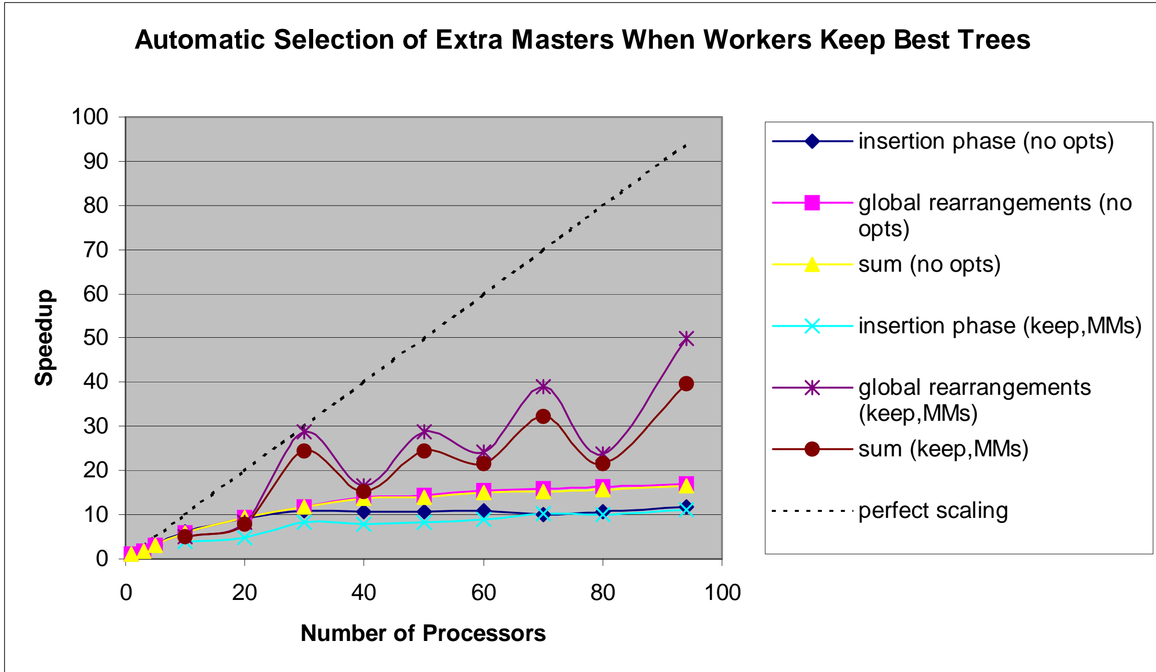


Figure 7.24: Speedup for automatic selection of extra masters combined with the option for workers to keep their best trees.

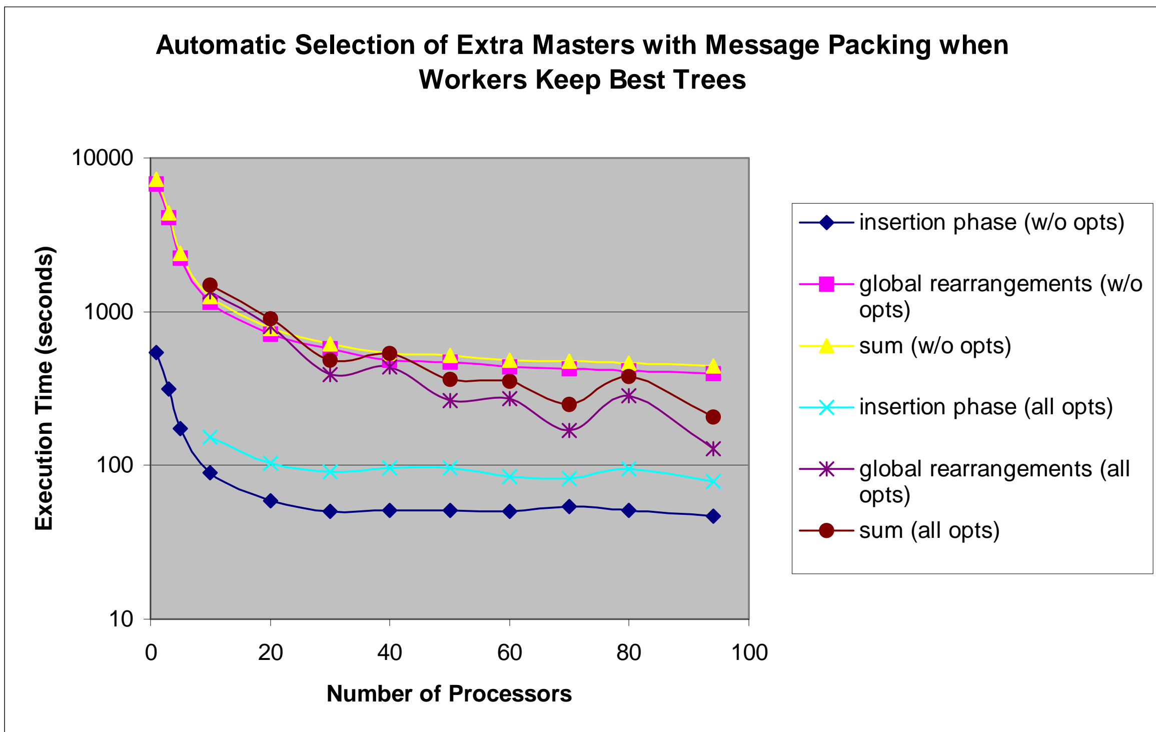


Figure 7.25: Execution times for automatic selection of extra masters combined with the options for workers to keep their best trees and message packing with a bundle size of 5.

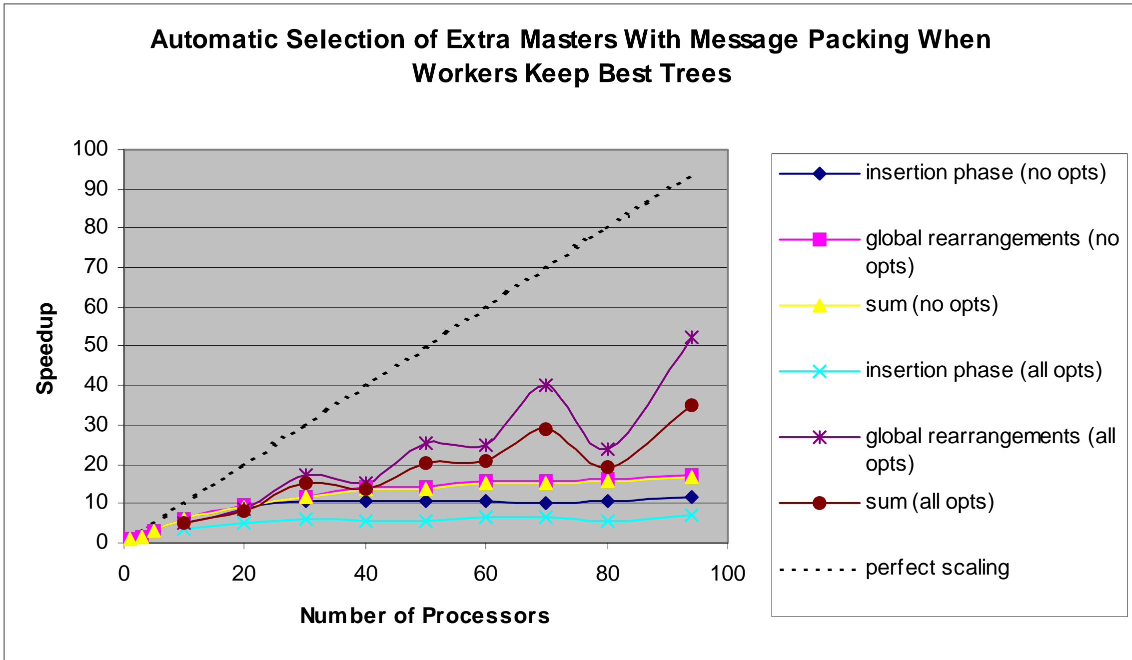


Figure 7.26: Speedup for automatic selection of extra masters combined with the options for workers to keep their best trees and message packing with a bundle size of 5.

The effect on performance of these two combinations is compared with the effects of the optimizations individually in Figure 7.27 below. The speedup obtained in the absence of any optimizations is provided as a reference.

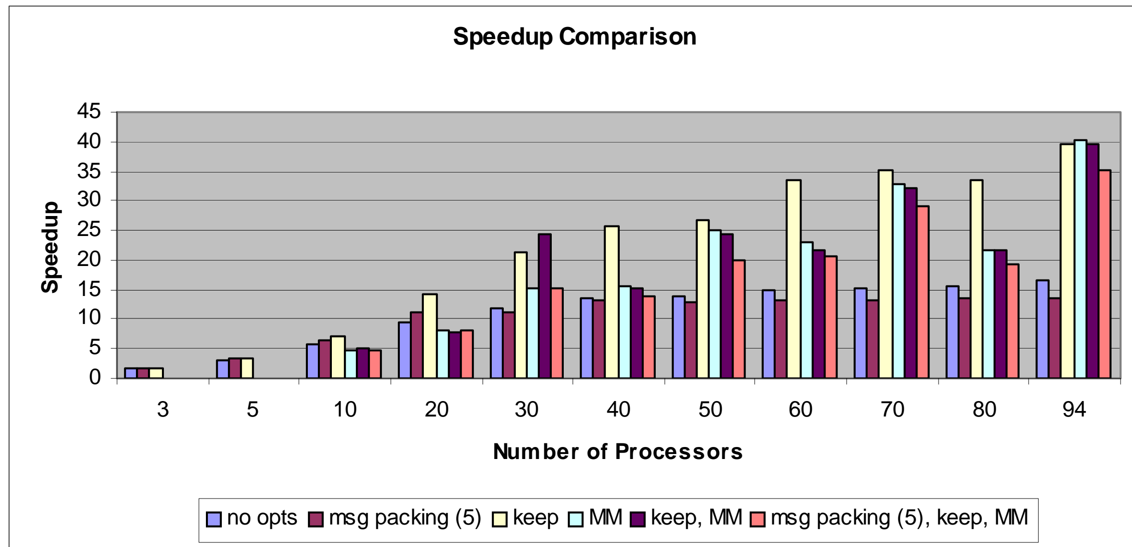


Figure 7.27: Speedup comparison for individual optimizations and in combination. Speedup was determined from the sum of the insertion phase and global rearrangements. Automatic selection of the number of extra masters is in effect.

7.7 Achieving Results Identical to fastDNAml

As stated previously, one of the goals of this work was to produce the same trusted results as fastDNAml, both in topology and numerically, but in a shorter amount of time by taking advantage of optimized parallelization. This goal was achieved, as identical results were produced by both programs under all conditions that were tested. Thus, of the 49,114 topologies evaluated, both programs reported the same topology, with the same likelihood and branch lengths. Several efforts had to made, as discussed previously in Section 4.4.1, in order for the program developed to arrive at the same results as fastDNAml, including adjustments in the precision used in tree message string representation, as well as allowing the master to perform some initializations of the branch lengths following topology generation. It should be noted that while both programs produce the same results, a limited

number of decimal places are printed to the output, which may hide slight differences in values between the two programs.

The results achieved with the optimizations designed and implemented in this work suggest their potential advantages when applied to large-scale systems and problems. This is where the bottleneck will be most prominent, hindering the performance of existing works using the master/worker scheme. The use of multiple masters and workers keeping their best trees both showed a dramatic improvement in performance over the initial parallelization. Their combination, however, did not result in any further improvement. Under the conditions analyzed, message packing was shown to slightly reduce performance, but the results suggest that significant performance gains may be achievable for large-scale problems. The generation of results identical to that of fastDNAml demonstrates that accuracy is not comprised with parallelization or its subsequent optimization. Chapter 8 discusses what has been gained from these demonstrated benefits of the optimizations, and what they leave for the future.

Chapter 8 Conclusions

Demonstration of the effects on performance from each of the optimizations, as provided in the previous chapter, marks the last major step for this thesis work. Now there is a need to reflect on what has been accomplished, and how this work contributes to the field. With many questions now answered, many more have arisen. Much work thus remains for the future, as several difficulties were encountered which put a limit on what could be completed within a reasonable amount of time. This chapter summarizes the accomplishments of this thesis, and highlights directions for possible future work.

8.1 Recapitulation

For this work the popular sequential fastDNAml program for maximum likelihood-based phylogenetic inference was parallelized in order to demonstrate the effectiveness of three optimizations developed specifically for the masters/workers parallelization approach. Initial background research revealed that previous parallelization attempts, which typically use the master/workers scheme, are inadequate when applied to large-scale systems and problems due to a communication bottleneck arising from the use of a single master. Not only are large numbers of workers available with supercomputers, but practical phylogenetic inference problems may involve the analysis of thousands of taxa, with the number of potential topologies in the search space growing exponentially with the number of taxa. Although the maximum likelihood method utilizes heuristics to reduce the number of topologies, a huge number still remains, and when coupled with the computational intensity of the maximum likelihood method, the need for parallelization is apparent. While

a few innovative algorithms targeted for larger problem sizes have been developed recently, biologists are hesitant to use these unfamiliar parallel algorithms as they do not trust the results. The goal of this work was to demonstrate several ways in which the more familiar master/workers approach can be optimized to make execution with large-scale systems and problems feasible. In order for biologists to be more likely to embrace these unfamiliar optimizations, the sequential algorithm of fastDNAml was used as a foundation, and results identical to fastDNAml were obtained following parallelization and optimization. fastDNAml is one of the less complex phylogenetic inference programs available based on the maximum likelihood method, which made it an ideal foundation. Other popular parallelizations based on fastDNAml exist, which use the master/workers approach. These programs also use more advanced algorithms for calculating branch lengths, likelihood, etc., which are claimed to lead to more accurate results in less time than those used by the simpler fastDNAml. It is the hope that the optimizations developed for this thesis will eventually be incorporated into these more-advanced works.

In order to demonstrate the effectiveness of the optimizations, fastDNAml was first parallelized for this work using the master/workers scheme. The branch length and likelihood calculations for individual trees are computed in parallel by the workers, as this is the most computationally intensive step in the process. Results were obtained demonstrating the presence of a bottleneck, as the speedup was not shown to increase significantly with the number of processors. Thus, the performance potentially gained from more available processors is limited. The message packing option was then implemented, whereby multiple trees needing evaluation are bundled into a single message, rather than sending out each tree in an individual message. The reduced overheads at the master were

expected to lead to a performance gain. Unfortunately, this was not seen from the results, which instead showed a slight reduction in performance for all bundle sizes tested. The results did suggest, however, that if more trees were evaluated, a performance gain might be possible. When the optimization was implemented where workers keep the best trees that they evaluate, a dramatic improvement in performance resulted, as expected, since substantially fewer tree results required handling by the master. It appeared that the factor by which performance improved increased with the number of processors. The results also suggested that an even greater improvement might be seen if more trees were evaluated. The third optimization, multiple masters, was designed to parallelize the responsibilities of the master. Dramatic improvement was seen here as well, with the factor by which the speedup improved increasing with the number of processors. In addition, the results suggested that the performance would further improve with an increase in the number of trees. Functionality for automatic selection of the number of additional masters was also included and is based on the total number of processors selected. It was seen from the results, however, that the selection mechanism requires refinement in order to take full advantage of the benefits gained through multiple masters. Even with the dramatic improvement seen with two of the three optimizations, the speedup still fell short of the desired near perfect scaling. Aside from the results obtained relating to execution times, program results relating to topology were also generated, which were found to be identical to those obtained with fastDNAml.

The results presented in this thesis suggest that performance gains associated with the optimizations have the potential to become significant when large-scale systems and problems are encountered. It is under these conditions that existing parallel efforts are

limited due to the communication bottleneck that arises with the master/workers scheme. Owing to the optimizations introduced, it is thought that this work will outperform existing efforts by reducing the bottleneck. Unfortunately, this could not be verified as only a limited number of processors were available and time constraints prevented analysis of large numbers of taxa.

8.2 Difficulties

Not uncommon for thesis work, difficulties were encountered at many stages of process that needed to be worked around. As discussed in the previous section, results for large-scale systems and problems could not be generated for several reasons. An attempt was made to obtain the sequential execution time for a dataset of 72 taxa, as opposed to 50, but on each of the several attempts, the connection to the cluster was lost. This was believed to result from time limitations associated with the method of accessing the cluster rather than from a problem with the program itself. Another error encountered while generating results was an insufficient amount of memory requested through P4_GLOBMEMSIZE when message packing was in effect, as discussed in Section 5.1.1. Although increasing this value fixed the problem for the conditions analyzed, it may reappear at a larger-scale.

Difficulties were also faced at earlier stages of the thesis. For instance, due to the complexity of the maximum likelihood method, the initial plan to implement the program entirely from scratch was abandoned. Even with a foundation code, however, development proved difficult, as the relatively simple fastDNAml program consisted of over 75 pages of scarcely commented code, with very little associated documentation. Numerous comments have since been added, both for code portions specific to fastDNAml as well as the

incorporated modifications. This should facilitate introduction of any future improvements. In addition, no one was available locally with a familiarity with the implementation of the program or of the maximum likelihood method. Also, as the work encompassed several disciplines, detailed advice was difficult to find, as those available were generally familiar with either the biological or computational side of the problem, but not both. Thus, much time was lost deciphering the fastDNAml code before parallelization could be implemented. A good understanding of the foundation code was necessary in order to integrate the parallelization into it. Another difficulty associated with using a foundation code rather than implementing from scratch was the need to work around the existing implementation. The node representation used for trees in fastDNAml, as shown in Figure 3.1, provides a good example of this, as the representation made it too difficult to reference specific nodes over multiple processors. This node representation also led to the specific implementation used for workload distribution over multiple masters.

The limited amount of time available to complete this thesis work, a good deal of which was lost to gaining an understanding of the foundation code, also affected the amount of work accomplished. Many interesting aspects of the results would have been explored further if time permitted, along with the generation of results with varied testing conditions. Incorporating additional optimizations into the program developed was also intended, but with the time constraints these ideas were set aside for the future.

8.3 Future Work

Potential future investigations have been alluded to on several occasions in this document. These possibilities will now be elaborated on, and include:

Large-scale systems and problems: This work was designed with the intention of making parallelization using the master/workers approach feasible with large-scale systems and problems. Therefore, an appropriate next step would be to gain access to a supercomputer and obtain results for a large dataset, containing possibly thousands of taxa. It would also be informative to run other existing parallel maximum likelihood-based phylogenetic inference programs, which use the master/workers approach, with large-scale systems and problems. Results could then be compared to see if they are outperformed by the optimized parallelization developed in this work. Results obtained for pfastDNAml, the official parallelization of fastDNAml, would be an especially interesting comparison with this work. From inspection of the pfastDNAml code, it does not appear that pfastDNAml would produce results identical to fastDNAml, based on the problems encountered in this work.

Alternative implementations for multiple masters: Of the three optimizations developed, multiple masters is believed to offer the most potential for achieving scalability with system and problem size. This is because the master's responsibilities are parallelized in addition to tree evaluation, which would otherwise result in a restrictive bottleneck. Many different designs could be used to implement multiple masters, with only one explored for this work. For instance, multiple layers of extra masters could be employed, assuming sufficient numbers of processors are available. This would allow the

responsibilities of each extra master in a layer to be parallelized through multiple extra masters in the next layer. Ideally, the number of layers would be designed to scale with the system and problem size. The appropriate number of extra masters, and possibly layers, could be changed as the program progresses, rather than selected on startup, by evaluating the conditions present at any given time, such as the number of trees to be evaluated. Reductions in performance caused by too few trees or processors could be avoided by not using any extra masters under these conditions. A starting point would be to refine the current automatic selection mechanism by determining what factors influence the number of extra masters that yields the best performance. The existing functionality already in place should facilitate the incorporation of any enhanced features.

Support for heterogeneous clusters: The program developed could also be further improved through the addition of functionality for responding to the varied processor speeds seen with heterogeneous clusters. In such a cluster, different workers may run on processors with different speeds. Thus, from one worker to the next the wall-clock time required to evaluate an individual tree would be likely to vary. The current program is designed under the assumption that each worker executes at the same speed, as the master aims to send equal numbers of trees to each of the workers. With a heterogeneous cluster, a faster processor is likely to complete evaluation of all its assigned trees long before a slower processor, and thus it will sit idle until the next round. The full benefits of parallelization would therefore not be achieved. A similar situation could occur with a homogenous cluster if some processors are more heavily loaded than others with outside work. One such way in which to remedy this situation would be to have the master occasionally receive a value from each worker indicating the average wall-clock time taken

to evaluate a tree. When a difference in time arises between workers, the master would adjust the number of trees sent to each worker by the appropriate factor. The efficiency of parallelization would thus improve, as processors available to perform evaluations would be less likely to sit idle.

Package of parallel programs: It is the aim that the program developed for this work will eventually be incorporated into a package of parallel programs offering phylogenetic inference with different methods. This would allow biologists to compare the results obtained from several methods easily, with parallelization allowing results to be obtained within a practical amount of time. Programs offering the maximum parsimony method and the disk-covering method, which may be applied to any of the methods of phylogenetic inference, are currently under development within the research group.

Bibliography

[1] Altekar, G., Dwarkadas, S., Huelsenbeck, J.P., and Ronquist, F. Parallel Metropolis coupled Markov chain Monte Carlo for Bayesian phylogenetic inference. *Bioinformatics* 20 (2004): 407-415.

[2] Bader, D.A., Chandu, V.P., and Yan, M. ExactMP: An Efficient Parallel Exact Solver for Phylogenetic Tree Reconstruction Using Maximum Parsimony. International Conference on Parallel Processing (ICPP'06), 2006, pp. 65-73.

[3] Bader, D., Moret, B., and Vawter, L. Industrial applications of high-performance computing for phylogeny reconstruction. *Proc. of SPIE ITCom* 4528 (2001): 159-168.

[4] Bodlaender, H., Fellows, M., and Warnow, T. Two strikes against perfect phylogeny. *Proceedings of the 19th International Colloquium on Automata, Languages, and Programming,* Lecture Notes in Computer Science, Springer-Verlag, NY, 1992, pp. 273-283.

[5] Chor, B. and Tuller, T. Maximum likelihood of evolutionary trees is hard. *Proc. of RECOMB05* (2005).

[6] Du, Z., Stamatakis, A., Lin, F., Roshan, U. and Nakhleh, L. Parallel divide-and-conquer phylogeny reconstruction by maximum likelihood. *Proceedings of the 2005 International Conference on High Performance Computing and Communications (HPCC 05),* 2 (2005): 346-350.

[7] Felsenstein, J. The number of evolutionary trees. *Syst. Zool.* 27 (1978): 27-33.

[8] Felsenstein, J. Evolutionary trees from DNA sequences: A maximum likelihood approach. *J. Mol. Evol.* 17 (1981): 368-376.

[9] Felsenstein, J. Models of DNA Evolution, Ch. 13 from *Inferring Phylogenies*. Sunderland, MA: Sinauer Associates, Inc., 2004.

[10] Felsenstein, J. PHYLIP. Accessed July 2007. http://evolution.genetics.washington.edu/phylip.html

[11] Felsenstein, J. and Churchill, G. A Hidden Markov Model Approach to Variation Among Sites in Rate of Evolution. Molecular Biology and Evolution 13 (1996): 93-104.

[12] GenBank Overview. Accessed Oct. 2006. http://www.ncbi.nlm.nih.gov/Genbank/index.html.

[13] Guindon, S. and Gascuel, O. A simple, fast, and accurate algorithm to estimate large phylogenies by maximum likelihood. *Syst. Biol.* 52 (2003): 696-704.

[14] Holder, M. and Lewis, P.O. Phylogeny estimation: traditional and Bayesian approaches. *Nat Rev Genet.* 4 (2003): 275-284.

[15] Howard, C. Parallelization of a Maximum Parsimony Branch and Bound Algorithm for Phylogenetic Inference. Master's thesis, Rochester Institute of Technology, 2005.

[16] Hu, J-M., M. Lavin, M. F. Wojciechowski, and M. J. Sanderson. Phylogenetic analysis of nuclear ribosomal ITS/5.8 S sequences in the tribe Millettieae (Fabaceae): *Poecilanthe-Cyclolobium*, the core Millettieae, and the *Callerya* group. *Systematic Botany* 27 (2002): 722-733.

[17] Huelsenbeck, J.P., Larget, B., van der Mark, P., and Ronquist, F. MrBayes: Bayesian Inference of Phylogeny. Accessed Oct. 2006. http://mrbayes.csit.fsu.edu/

[18] Huelsenbeck, J.P. and Ronquist, F. MRBAYES: Bayesian inference of phylogenetic trees. *Bioinformatics* 17 (2001): 754-755.

[19] Huson, D., Nettles, S., Parida, L., Warnow, T. and Yooseph, S. The Disk-Covering Method for Tree Reconstruction. *Proceedings of the Workshop on Algorithms and Experiments (ALEX),* Trento, Italy, 1998.

[20] Kishino, H. and Hasegawa, M. Evaluation of the Maximum Likelihood Estimate of the Evolutionary Tree Topologies from DNA Sequence Data, and the Branching Order in Hominoidea. *Journal of Molecular Evolution* 29 (1989): 170-179.

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

[22] MPICH. Accessed Aug. 2007. http://www-unix.mcs.anl.gov/mpi/mpich1/

[23] Olsen, G.J., Matsuda, H., Hagstrom, R., and Overbeek, R. fastDNAml: A tool for construction of phylogenetic trees of DNA sequences using maximum likelihood. *Comput. Appl. Biosci.* 10 (1994): 41-48.

[24] Olsen, G. and Overbeek, R. User documentation for fastDNAml version 1.2. Accessed Aug. 2007. http://www.bioinfo.hku.hk/fastDNAml_doc_1.2.txt

[25] Prion Deletion Mutants. Accessed Oct. 2006. http://www.mad-cow.org/prion_repeat_deletions.html

[26] RIT Research Computing Cluster. Accessed Oct. 2006. http://cluster.rit.edu/

[27] Ronquist, F. and Huelsenbeck, J.P. MRBAYES 3: Bayesian phylogenetic inference under mixed models. *Bioinformatics* 19 (2003): 1572-1574.

[28] Roshan, U., Moret, B., Williams, T., and Warnow, T. Rec-I-DCM3: A Fast Algorithmic Technique for Reconstructing Large Phylogenetic Trees. *Proceedings of the IEEE Computational Systems Bioinformatics Conference,* Aug. 16-19, 2004, pp. 98-109.

[29] Saitou, N. and Nei, M. The Neighbor-Joining Method: A New Method for Reconstructing Phylogenetic Trees. *Mol. Biol. Evol.* 4 (1987): 406-425.

[30] Schmidt, H., Strimmer, K., Vingron, M., and von Haeseler, A. Tree-puzzle: maximum likelihood phylogenetic analysis using quartets and parallel computing. *Bioinformatics* 18 (2002): 502-504.

[31] Snell, Q., Whiting, M., Clement, M., and McLaughlin, D. Parallel Phylogenetic Inference. *Supercomputing, ACM/IEEE 2000 Conference*, Nov. 4-10, 2000, p 35.

[32] Stamatakis, A. An Efficient Program for Phylogenetic Inference Using Simulated Annealing. *Proceedings of 19th IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS2005)*, Denver, Colorado, April, 2005.

[33] Stamatakis, A. Phylogenetic Models of Rate Heterogeneity: A High Performance Computing Perspective. *Proceedings of 20th IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS2006)*, Rhodos, Greece, April, 2006.

[34] Stamatakis, A. RAxML-VI-HPC: Maximum Likelihood-based Phylogenetic Analyses with Thousands of Taxa and Mixed Models. *Bioinformatics* 22 (2006): 2688-2690.

[35] Stamatakis, A., Ludwig, T., and Meier, H. New Fast and Accurate Heuristics for Inference of Large Phylogenetic Trees. *Proceedings of 18th IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS2004)*, Santa Fe, New Mexico, April, 2004.

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

[37] Stamatakis, A., Ott, M., and Ludwig, T. RAxML-OMP: An Efficient Program for Phylogenetic Inference on SMPs. *Proceedings of 8th International Conference on Parallel Computing Technologies (PaCT2005),* Lecture Notes in Computer Science, Springer-Verlag, 2005, pp. 288-302.

[38] Stewart, C.A., Hart, D., Berry, D.K., Olsen, G.J., Wernert, E.A., and Fischer, W. Parallel implementation and performance of fastDNAml: a program for maximum likelihood phylogenetic inference. *Proc. 2001 ACM/IEEE Conference on Supercomputing (CDROM)*, ACM, 2001, pp. 32.

[39] Strimmer, K. and von Haeseler, A. Quartet puzzling: a quartet maximum-likelihood method for reconstructing tree topologies. *Mol. Biol. Evol*. 13 (1996): 964-969.

[40] Studier, J. and Keppler, K. A note on the neighbor-joining algorithm of Saitou and Nei. *Molecular Biology and Evolution* 5 (1988): 729-731.

[41] Swofford, D. PAUP. Accessed Oct. 2006. http://paup.csit.fsu.edu/

[42] Swofford, D., Olsen, G., Waddell, P. and Hillis, D. Phylogenetic Inference, Ch. 11 from *Molecular Systematics, 2nd ed*. Ed. Hillis, D., Moritz, C. and Mable, B. Sunderland, MA: Sinauer Associates, Inc., 1996.

[43] TreeBASE, A Database of Phylogenetic Knowledge. Accessed Aug. 2007. http://www.treebase.org

[44] Unofficial versions. Accessed Oct. 2006. http://atgc.lirmm.fr/phyml/versions.html.

[45] Vinh,L.S. and von Haeseler,A. IQPNNI: Moving fast through tree space and stopping in time. *Mol. Biol. Evol.* 21 (2004): 1565–1571.

[46] Whelan, S., Lio, P., and Goldman, N. Molecular Phylogenetics: State-of-the-Art Methods for Looking into the Past. *Trends in Genetics* 17 (2001).

[47] Williams, T., Berger-Wolf, B. Roshan, U., and Warnow, T. The relationship between maximum parsimony scores and phylogenetic tree topologies. Tech. Report, TR-CS-2004-04, Dept. of Computer Science, The University of New Mexico, 2004.

[48] Williams, T.L. and Moret, B.M.E. An Investigation of Phylogenetic Likelihood Methods. *Bioinformatics and Bioengineering, 2003. Proceedings. Third IEEE Symposium,* March 10-12, 2003, pp.79-86.

[49] Zhou, B., Till, M., Zomaya, A. and Jermiin, L. Parallel Implementation of Maximum Likelihood Methods for Phylogenetic Analysis. *Parallel and Distributed Processing Symposium, 2004. Proceedings, 18th International.* April 26-30, 2004, p 237.

[50] Zwickl, D. J. Genetic algorithm approaches for the phylogenetic analysis of large biological sequence datasets under the maximum likelihood criterion. Ph.D. dissertation, The University of Texas at Austin, 2006.

[51] Zwickl, D.J. GARLI. Accessed Nov. 2006. www.bio.utexas.edu/faculty/antisense/garli/Garli.html