Parallelization of a Maximum Parsimony Branch and Bound Algorithm for Phylogenetic Inference

Curtis J. Howard

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
PARALLELIZATION OF A MAXIMUM PARSIMONY BRANCH AND BOUND ALGORITHM FOR PHYLOGENETIC INFERENCE

By

Curtis J. Howard

January 2006

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Engineering

Rochester Institute of Technology

Approved by:

Muhammad Shaaban

Primary Advisor: Dr. Muhammad Shaaban, Assistant Professor

Larry Buckley

Advisor: Dr. Larry Buckley, Bioinformatics Professor

Juan Carlos Cockburn

Advisor: Dr. Juan Carlos Cockburn, Associate Professor
THESIS RELEASE PERMISSION FORM

Rochester Institute of Technology
Kate Gleason College of Engineering

Title: Parallelization of a Maximum Parsimony Branch and Bound Algorithm for Phylogenetic Inference.

I, Curtis Howard, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or in part.

Curtis Howard

Date
Dedication

This work is dedicated to my parents who taught me the importance of commitment and perseverance (and coffee) in whatever it was that I was trying to achieve.
Acknowledgements

The author would like to thank Dr. Muhammad Shaaban, Dr. Larry Buckley and Dr Juan Cockburn for participating in the author's thesis committee. The author would also like to thank the Rochester Institute of Technology and in particular the Computer Engineering Department of the Kate Gleason College of Engineering and its entire faculty for the exemplary job it has done in preparing the author for the research work required by this thesis.
Abstract
PARALLELIZATION OF A MAXIMUM
PARSIMONY BRANCH AND BOUND
ALGORITHM FOR PHYLOGENETIC
INFERENC

by Curtis Howard

Supervising Professor: Dr. Muhammad Shaaban
Department of Computer Engineering

Phylogenetic inference involves the reconstruction of evolutionary relationships among species in the form of branching diagrams called trees. Specifically, certain biological structures common to all living organisms, such as morphological characteristics, protein sequences or DNA sequences can be compared. Differences and similarities in these characteristics among species are used to reconstruct the evolutionary relationships and draw trees. Many methods of tree reconstruction are currently used. The method of maximum parsimony for phylogenetic inference is a widely used algorithm which employs the hypothesis that the most likely tree for a given group of data will be the one which uses the least number of changes from an origin (root of the tree) to the terminal taxa. The problems and corresponding solution algorithms associated with these searches are frequently implemented on single-processor systems, and can take weeks to complete for large data sets.
Parallelization of these algorithms is therefore an important area of development in the bioinformatics community [1, 3, 17, 20, 25]. A free license, open-source, parallel implementation of a phylogenetic inference program using maximum parsimony has yet to be developed, and it is the aim of this thesis to provide such a tool. It is hoped that the tool will work transparently with one of the most popular suites of free phylogenetic inference tools called PHYLIP, developed by Joe Felsenstein at the University of Washington [7], by accepting and generating the same format of input and output data. The tool would be a first step towards providing the academic community and others with improvements in performance and capabilities (through parallelization) over the currently available free distributions of phylogenetic inference programs using parsimony, allowing for larger volumes of data to be analyzed in a reduced amount of time.
# TABLE OF CONTENTS

Dedication .................................................................................................................. iii
Acknowledgements ...................................................................................................... iv
Abstract ......................................................................................................................... v
Glossary ......................................................................................................................... x

Chapter 1 - Introduction ............................................................................................... 1
  1.1 Goals ...................................................................................................................... 2
  1.2 Chapter Outline ..................................................................................................... 3

Chapter 2 - Motivation ................................................................................................. 5
  2.1 Phylogenetic Analysis ............................................................................................. 5
  2.2 Previous Work & Feasibility .................................................................................. 6
  2.3 Possible Uses ......................................................................................................... 12

Chapter 3 - Maximum Parsimony ............................................................................... 14
  3.1 The Data ............................................................................................................... 15
  3.2 The Algorithm ....................................................................................................... 16
    3.2.1 Weighted vs. Un-weighted ............................................................................. 16
    3.2.2 Rooted vs. Un-rooted .................................................................................... 17
    3.2.3 The Fitch Parsimony Algorithm .................................................................... 20
  3.3 Pre-Scoring Optimizations ..................................................................................... 22

Chapter 4 - The Branch and Bound Shortcut ............................................................... 25
  4.1 Tree Topology Representation ............................................................................... 26
    4.1.1 Tree Identifiers ............................................................................................. 26
    4.1.2 Tree Generation and the Newick Format ...................................................... 28
  4.2 Branch and Bound .................................................................................................. 29

Chapter 5 - Parallelization of the Search .................................................................... 34
  5.1 Characterization of the Problem ............................................................................ 34
  5.2 Previous Work ....................................................................................................... 37
  5.3 Solution Considerations ......................................................................................... 37
    5.3.1 Tree Topology Distribution Strategy ............................................................ 38
    5.3.2 Grain Size ..................................................................................................... 38
    5.3.3 Threshold Updating ....................................................................................... 39

Chapter 6 - Implementation ........................................................................................ 41
  6.1 Module Summary .................................................................................................... 41
  6.2 Input Data Format .................................................................................................. 45
  6.3 Initialization and Pre-Score Optimization ............................................................... 47
  6.4 The score and treegen Modules ............................................................................ 48
    6.4.1 The score Module ......................................................................................... 48
    6.4.2 The treegen Module ..................................................................................... 49
  6.5 The treedD and worker Modules ......................................................................... 50
<table>
<thead>
<tr>
<th>Number</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1: Possible labeled, rooted, bifurcating trees for 2, 3, and 4-taxa</td>
<td>8</td>
</tr>
<tr>
<td>Figure 2.2: Number of n-taxa rooted, bifurcating trees that exist</td>
<td>9</td>
</tr>
<tr>
<td>Figure 2.3: Search space sizes for varying taxa in rooted, bifurcating tree problems</td>
<td>9</td>
</tr>
<tr>
<td>Figure 3.1: Sample input data for a typical phylogenetic program</td>
<td>16</td>
</tr>
<tr>
<td>Figure 3.2: Sample cost matrix for a weighted parsimony algorithm</td>
<td>17</td>
</tr>
<tr>
<td>Figure 3.3: A 4-taxa unrooted tree and the corresponding rooted trees it represents</td>
<td>18</td>
</tr>
<tr>
<td>Figure 3.4: Number of n-taxa un-rooted, bifurcating trees that exist</td>
<td>20</td>
</tr>
<tr>
<td>Figure 3.5: Fitch parsimony algorithm for 1* column of fig 3.1 sample input data</td>
<td>21</td>
</tr>
<tr>
<td>Figure 3.6: Scored input data matrix for the tree topology in figure 3.5</td>
<td>24</td>
</tr>
<tr>
<td>Figure 4.1: Tree identifier vector example</td>
<td>27</td>
</tr>
<tr>
<td>Figure 4.2: The Newick string tree topology representation format</td>
<td>29</td>
</tr>
<tr>
<td>Figure 4.3: Illustration of Branch and Bound</td>
<td>33</td>
</tr>
<tr>
<td>Figure 6.1: Flowchart overview of program execution and used modules</td>
<td>44</td>
</tr>
<tr>
<td>Figure 6.2: Two valid and equivalent 5-taxa input data files</td>
<td>46</td>
</tr>
<tr>
<td>Figure 6.3: Example of Fitch parsimony reduction for a given site in the search module</td>
<td>49</td>
</tr>
<tr>
<td>Figure 6.4: mli_proc module parallel search algorithm pseudo-code</td>
<td>56</td>
</tr>
<tr>
<td>Figure 7.1: Distribution strategy option 1 -- Barrier</td>
<td>59</td>
</tr>
<tr>
<td>Figure 7.2: Distribution strategy option 2 -- dynamic</td>
<td>60</td>
</tr>
<tr>
<td>Figure 7.3: Loss of efficiency when the root node does no searching</td>
<td>61</td>
</tr>
<tr>
<td>Figure 7.4: Parallelization Architecture vs. Node Utilization</td>
<td>62</td>
</tr>
<tr>
<td>Figure 7.5: Grain Size (taxa-level) vs. Performance (Execution time)</td>
<td>63</td>
</tr>
<tr>
<td>Figure 7.6: Threshold Update Interval vs. Performance (Execution Time)</td>
<td>64</td>
</tr>
<tr>
<td>Figure 7.7: Sample Input Data</td>
<td>65</td>
</tr>
<tr>
<td>Figure 7.8: Output data corresponding to input in figure 7.7</td>
<td>66</td>
</tr>
<tr>
<td>Figure 7.9: Speedup for 9-taxa</td>
<td>68</td>
</tr>
<tr>
<td>Figure 7.10: Speedup for 11-taxa</td>
<td>69</td>
</tr>
<tr>
<td>Figure 7.11: Speedup for 11-taxa on the 30-node cluster of VLSI workstations</td>
<td>70</td>
</tr>
<tr>
<td>Figure 7.12: Execution time on VLSI workstations vs. the safault cluster</td>
<td>71</td>
</tr>
<tr>
<td>Figure 8.1: Conversion of an integer to a vector identifier used in Treegeen</td>
<td>74</td>
</tr>
<tr>
<td>Figure 9.1: 11-taxa, 18 character input file with random data</td>
<td>78</td>
</tr>
<tr>
<td>Figure 9.2: 17-taxa, 18 character input file with simple ordered data</td>
<td>78</td>
</tr>
<tr>
<td>Figure A.1: The GUI when it is first opened</td>
<td>88</td>
</tr>
<tr>
<td>Figure A.2: An example of the GUI after it has finished a search</td>
<td>89</td>
</tr>
<tr>
<td><strong>Glossary</strong></td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td><strong>Barrier Parallelization</strong></td>
<td>A method of parallelization in which nodes of the parallel computer wait for others to arrive at a certain point in execution before continuing.</td>
</tr>
<tr>
<td><strong>Bifurcating</strong></td>
<td>Refers to a point in a phylogenetic tree in which an ancestral taxon splits into two independent lineages. [4]</td>
</tr>
<tr>
<td><strong>Branch &amp; bound</strong></td>
<td>A means of streamlining the search for trees of maximum parsimony. The algorithm builds trees one taxa at a time, and eliminates entire groups of trees from the scoring process on the basis of predictive knowledge that their scores will be greater than the best score found so far.</td>
</tr>
<tr>
<td><strong>Communication-to-Computation Ratio</strong></td>
<td>A metric used in parallel computer architectures, which shows the amount of time nodes in the parallel computer spend communication versus computing the result of the problem at hand.</td>
</tr>
<tr>
<td><strong>Dendrogram format</strong></td>
<td>A graphical representation of a tree topology, showing nodes and branches through a network of connected lines.</td>
</tr>
<tr>
<td><strong>DNA</strong></td>
<td>Deoxyribonucleic Acid. A usually double-stranded biopolymer of linked nucleotides in which the sugar residue is deoxyribose. The molecular basis of heredity.</td>
</tr>
<tr>
<td><strong>Evolutionary theory</strong></td>
<td>The process by which all forms of plant and animal life change slowly over time because of slight variations in the genes that one generation passes down to the next. [4]</td>
</tr>
<tr>
<td><strong>Fitch parsimony</strong></td>
<td>A scoring algorithm that implements the parsimony criteria for un-rooted trees only.</td>
</tr>
<tr>
<td><strong>GUI</strong></td>
<td>Graphical User Interface.</td>
</tr>
<tr>
<td><strong>Heuristic search</strong></td>
<td>Non-deterministic method of searching for phylogenetic trees.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Identifier vector (tree)</td>
<td>An array of numbers used internally by the implementation of the search program developed in this thesis, to represent the topology of a specific un-rooted phylogenetic tree.</td>
</tr>
<tr>
<td>Message Passing Cluster</td>
<td>A group of networked computers, cooperating to run programs solving a common problem by sharing information through messages sent across the network.</td>
</tr>
<tr>
<td>MPI</td>
<td>(Message Passing Interface) A computer communications protocol. It is a de facto standard for communication among the nodes running a parallel program on a distributed memory system.</td>
</tr>
<tr>
<td>Multifurcating tree</td>
<td>A graphical representation of an unknown branching order involving 3 or more species in a phylogenetic tree. [4]</td>
</tr>
<tr>
<td>Newick format</td>
<td>In a computer program, the format in which basic information about the structure of a phylogenetic tree is conveyed in a series of nested parentheses. [4]</td>
</tr>
<tr>
<td>NOW</td>
<td>(Network of Workstations) A computer network which connects several computer workstations together, and by utilizing special software it allows to use the network as a cluster</td>
</tr>
<tr>
<td>Parsimony</td>
<td>“Economy in the use of means to an end.” The process of attaching preference to one evolutionary pathway on the basis of which pathway requires the invocation of the smallest number of mutational events.[4]</td>
</tr>
<tr>
<td>Phylogenetics</td>
<td>A field of biology which studies the relationships between 3 or more genes or organisms.</td>
</tr>
<tr>
<td>Reduced-taxa tree</td>
<td>A term used in this thesis to describe a phylogenetic tree topology containing fewer taxa than the number specified in the input to the search (represented by an identifier vector).</td>
</tr>
<tr>
<td>Rooted tree</td>
<td>Phylogenetic tree in which a single node is designated as a common ancestor, and a unique path leads from it through evolutionary time to any other node. [4]</td>
</tr>
</tbody>
</table>
| Score (tree)                              | The number of mutations (calculated using a scoring algorithm such as Fitch parsimony) necessary for a given tree topology to
represent the given input data of the search.

**Taxa-level**
A term used in this thesis representing the number of taxa in a given reduced-taxa tree.

**Taxon**
A classificatory group of any rank. (e.g. family, genus, species, subspecies) Plural: taxa. For the purposes of this thesis, a taxon is a single line of input including a string of nucleotides which will be assigned to a specific leaf node on a phylogenetic tree.

**Threshold score**
A term used in this thesis representing the score of the most parsimonious tree(s) found so far. It is used in the branch and bound algorithm to determine which trees can be eliminated from the scoring process.

**Topology**
The structural features of a phylogenetic tree.

**TreeGen**
Software developed by [28] for the purpose of generating all possible un-rooted tree topologies for use in phylogenetic inference.

**Un-rooted tree**
Phylogenetic tree that specifies the relationship among nodes, but does not make any representation about the direction in which evolution occurred. [4]
Recent advances in computational power and technology over the past half-century have given rise to a number of new fields centered on the availability of the new resources that have spawned as a result. One such area of research is evolutionary biology, in which possible models for the evolution of various species are no longer hypothesized based on the physical attributes of the organisms in question, but rather systematically searched for through algorithms implemented on computers. The results of such methods have been shown to be much more accurate and effective than any previous ways, not only in determining which species branched from others, but also in estimating the times at which these events might have taken place. Maximum parsimony is one of the most widely accepted and prolific of the algorithms used for such a purpose, and was pioneered by Walter Fitch [8]. Essentially, it is based on the assumption that the most likely path of evolution will involve the least number of changes throughout time. Of course this ignores many factors including environment, geographic separation, diseases, and other complications; however, Fitch's maximum parsimony algorithms are still widely accepted and used today. Since the publication of this method of phylogenetic inference in 1971, many developments in the optimization of the algorithm have been suggested and implemented. The branch and bound method for instance, was proposed by Hendy and Penny in their 1982 paper [10]; depending on the dataset used for evaluation, it can provide
significant reduction in the search-space through elimination of entire groups of evolutionary trees from consideration by the parsimony algorithm.

1.1 Goals

Development of software implementing the algorithms discussed above has been primarily accomplished by bioinformatics researchers themselves. David Swofford's PAUP (Phylogenetic Analysis Using Parsimony) [21], and Joe Felsenstein's PHYLIP [7] are good examples. These programs have been accepted as the standard in both academic and industrial communities alike. Many other programs have also provided working solutions to the implementation of phylogenetic inference searches, but without as much acclaim as the above two. Finally, parallel versions of these programs have been developed [3, 17, 20, 25], but have been limited in scope. Many of them have been targeted towards specific inference algorithms, and do not offer the flexibility of some of the uni-processor programs [20, 17]. Others are intended for commercial use [3], and their source code is not available to the public for porting to new platforms; which is essential if many people are to use the software, since multiprocessor platforms are inherently different in composition. This is sometimes done intentionally, as manufacturers of MPPs (Massively Parallel Processors) will provide the binary versions of a parallel program, trying to coerce possible customers to purchase their machines. Finally, none of the parallel programs mentioned have both
allowed open-source access to the code, and implemented an evolutionary tree search using the maximum parsimony criteria outlined by [8]. It is the goal of this thesis to develop a tool that will meet these criteria. The program will implement the maximum parsimony algorithm described in [8] using the branch and bound search space reduction techniques described by [10], and will do so in a parallel manner through the use of the MPICH libraries for parallel programming. As well, by using the same expected input and output data formats that popular programs such as PAUP [21] and PHYLIP [7] use, the program developed in this thesis will hopefully inspire further research in the application of parallel computing to open-source phylogenetic inference using parsimony. Finally, the program will feature a GUI interface to simplify and encourage use of it. This GUI will be written in QT, a flexible and portable set of C++ libraries for cross-platform compatibility in GUI development.

1.2 Chapter Outline

The remainder of this document is organized as follows: Chapter 2 describes the motivation for this work, in terms of details of existing uni-processor and parallel evolutionary tree search software designs, and how this algorithm can provide a stepping stone to improve on them. In Chapter 3, the inner workings of phylogenetic inference using Fitch’s maximum parsimony, including pre-search optimizations, evolutionary tree scoring algorithms, and how this will be implemented, will be
covered. In Chapter 4, the branch and bound algorithm, and how it also will be implemented in this thesis, will be discussed; this will include the methods used to maintain knowledge of all trees that have been considered so far, allowing trees of specific desired topology to be formed. Chapter 5 will discuss the various options considered for the parallelization of the algorithm, and also the details of, and justifications for the one chosen. Chapter 6 will summarize the implementation of the algorithms described in chapters 3 through 5, in the prototype search program developed for this thesis. In chapter 7, the results of tests on several multi-processor platforms will be presented. This will include execution time, speedup, and comparisons between the different clusters the code was run on. In Chapter 8, the results and their meaning will be elaborated on. In Chapter 9, conclusions will be made regarding the results, and how well they met the goals of this thesis. In Chapter 10, possibilities for further improvements, and future paths to be taken from what I have done will be discussed. Finally, the Appendix will describe the system requirements of the program, and also how to install and run the program to obtain desired results. This will include notes on the restrictions the program imposes on input files, and also the possible forms of output, some of which are compatible with various tree viewing programs included in phylogenetic software suites such as PHYLIP [7].
Chapter 2 - Motivation

Evolutionary theory has been useful and even crucial to the advances we have seen in biology within the past century. It has provided scientists with a basis for understanding everything from drug resistance, to changes in virulent organisms, to explanations for the colors of peacock's tails [26]. Phylogenetic analysis in particular, has further provided a map relating these changes to one another, and has allowed many problems which seemed previously intractable to be solved. In this chapter, the usefulness of a high performance phylogenetic search using parsimony will be evaluated.

2.1 Phylogenetic Analysis

Before discussing the motives for parallelizing the parsimony branch & bound algorithms this thesis deals with, it would seem necessary to first justify the purpose of the algorithm itself. Phylogenetic analysis, which encompasses certain types of searches like parsimony, is a relatively new (within the past 50 years) field of biology which has revolutionized our methods of tracing ancestries, and roots for various biological entities. Discovery of drugs for example, has been facilitated through the examination of gene functions over time, and the ability to then predict future gene function from those results [26]. By examining the different types of mitochondrial DNA in humans using phylogenetic analysis, scientists have been able to trace human
history in the world geographically, and even find its roots [19]. The same process has
been adapted to tracing viruses such as HIV, and has allowed scientists to show that
the virus was transmitted to humans during the mid-twentieth century from
chimpanzees and mangabey monkeys [26]. It has been used to link convicted felons
with crimes [26], and can even be used to trace the sources of bioweapons [26]. Most
recently, efforts have been made to track protein folds, since they tend to be conserved
among diverging organisms with common ancestry. In fact, IBM has recently devoted
large amounts of its resources towards the development of a massively parallel
processor (MPP) called Blue-Gene (currently the most powerful computer in the
world) to be used for tracking of such protein folds [11].

2.2 Previous Work & Feasibility

As mentioned before, there is a plethora of commercial and freely available software
implementing most of the phylogenetic inference algorithms that exist, including
maximum parsimony. Most all of these distributions run on single processor
platforms. PHYLIP [7], written by Joe Felsenstein of the University of Washington,
and also PAUP [21], written by David Swofford of Florida State University, are by far
the most popular of these uni-processor programs. Both of these offer efficient
implementations of the maximum parsimony search. PAUP implements various types
of parsimony algorithms including an exhaustive branch & bound parsimony search;
however, it is closed source and must be purchased. PHYLIP offers dnapars, which is
a heuristic "uphill" parsimony search, and is open source. Other software such as MacClade [15], provide the user with a more graphically oriented, and interactive interface for more character analysis, but have weak search capabilities; and still others provide implementations of more obscure phylogenetic analysis algorithms.

The search space for these algorithms increases at a very fast rate as more species' DNA strings, referred to as taxa, are added to the input. This creates problems which are NP-Hard and NP-Complete [4], resulting in searches that can sometimes take weeks or longer to complete. Finding the most optimal trees in a deterministic phylogenetic algorithm involves taking into account every possible tree that could be formed with the data. Bifurcating trees, the most commonly used form, are trees in which each parent node splits into exactly two child nodes (or leaves). Knowing this fact, we can determine how many trees can be formed given a certain number of possible leaves, or taxa, which corresponds to the number of species DNA strings given as input to the algorithm. As shown in Figure 2.1 below, the number of possible trees to be formed using 3 instead of 2 taxa increases from 1 to 3, and from 4 instead of 3 taxa from 3 to 15.
If Figure 2.1 is continued for more taxa, the number of possible trees goes from 15 for 4 taxa, to 105 for 5 taxa, and so on. It can be shown that these numbers correspond to $1 \times 3 = 3$ trees for 3 taxa, $1 \times 3 \times 5 = 15$ for 4, and $1 \times 3 \times 5 \times 7 = 105$ for 5. Generalizing this, the $n^{th}$ taxon can be added in any one of $(2n - 3)$ places to create the possible $n$-taxa trees formed from a specific $n$-1 taxa tree. The equations in figure 2.2 represents the number of possible unrooted, bifurcating trees that exist for $n$-taxa. The second form is derived from the product form in the first.
\[ \text{trees}_{\text{rooted}} = 1 \times 3 \times 5 \times 7 \times \cdots \times (2n - 3) \]

\[ \text{trees}_{\text{rooted}} = \prod_{i=3}^{2n} (2n - 3) \]

\[ \text{trees}_{\text{rooted}} = \left[ \frac{(2n - 3)!}{2^{n-1}(n-1)!} \right] \]

Figure 2.2: Number of n-taxa rooted, bifurcating trees that exist. [6]

Using this formula, figure 2.3 gives search spaces for problems with varying numbers of taxa.

<table>
<thead>
<tr>
<th>Number of Taxa</th>
<th>Number of Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>105</td>
</tr>
<tr>
<td>6</td>
<td>945</td>
</tr>
<tr>
<td>7</td>
<td>10,395</td>
</tr>
<tr>
<td>8</td>
<td>135,135</td>
</tr>
<tr>
<td>9</td>
<td>2,027,025</td>
</tr>
<tr>
<td>10</td>
<td>34,459,425</td>
</tr>
<tr>
<td>11</td>
<td>654,729,075</td>
</tr>
<tr>
<td>12</td>
<td>13,749,310,575</td>
</tr>
<tr>
<td>13</td>
<td>316,234,143,225</td>
</tr>
<tr>
<td>14</td>
<td>7,905,853,580,625</td>
</tr>
<tr>
<td>15</td>
<td>213,458,046,676,875</td>
</tr>
<tr>
<td>16</td>
<td>6,190,283,353,629,375</td>
</tr>
<tr>
<td>17</td>
<td>191,898,783,962,510,625</td>
</tr>
<tr>
<td>18</td>
<td>6,332,659,870,762,850,625</td>
</tr>
<tr>
<td>19</td>
<td>221,643,095,476,699,771,875</td>
</tr>
<tr>
<td>20</td>
<td>8,200,794,532,637,891,559,375</td>
</tr>
<tr>
<td>30</td>
<td>4.9518 x 10^{53}</td>
</tr>
<tr>
<td>40</td>
<td>1.00985 x 10^{67}</td>
</tr>
<tr>
<td>50</td>
<td>2.75292 x 10^{76}</td>
</tr>
</tbody>
</table>

Figure 2.3: Search space sizes for varying taxa in rooted, bifurcating tree problems.
It should be evident by now that the rate at which the search space increases for a given increase in the number of taxa, is the reason behind which phylogenetic search problems are NP-hard and NP-complete.

Parallelization of these algorithms has been attempted before, but without as much acceptance from the bioinformatics community as uni-processor software such as PAUP, and PHYLIP. This may be in part due to the difficulty in obtaining cluster computing resources, and also to the lack of familiarity with cluster computing which may keep some biologists from exploring the area. If efforts have been made to parallelize phylogenetic inference software, they have primarily targeted heuristic versions of searches, which are an estimate of the most likely evolutionary tree and do not guarantee that the trees found are the optimal trees in the search space. Parallel programs such as fastDNAml [17], and RAxML [20] are two of the better known parallel search implementations; however, they use the Maximum Likelihood criteria, which is based on a different assumption about evolution than parsimony. Joe Felsenstein’s dnapars parsimony search will eventually be parallelized he says [7]; however, when and if it this does happen, it will still be a search based on the heuristic “uphill” search as opposed to a deterministic one discussed in this thesis. SGI [3], a high-end computer development company, has parallelized a few of the primary tools from PHYLIP including dnapars. Unfortunately, they are not disclosing the source files for this software, and only pre-compiled versions which will only run on their machines are available. The published results, including speedup graphs from their
program do however show that parsimony search algorithms can indeed be parallelized, and can offer great performance benefits over their uni-processor counterparts. As deterministic parsimony searches are very frequently used in phylogenetic analysis [2], it is surprising that parallelization of this type of search has not received much attention. Aside from SGI’s work, which is still a heuristic and non-deterministic search, no other significant attempts to parallelize a deterministic parsimony algorithm were found. An RIT graduate student in Computer Science made a first attempt at solving this problem in 2003 [12]; however, he was unable to obtain a program which returned correct results, and which could work for datasets containing DNA from more than 9 taxa. Both the research from SGI and from [12] show that the task at hand for this thesis is possible. Furthermore, algorithms involving Branch & Bound have also been shown to be good candidates for parallelization by [1]. Producing a market-competitive version of this parallel parsimony program would be out of the scope and available timeframe for this thesis. Therefore my primary goal is to develop an open-source program, employing the most efficient parallelization techniques to provide scalability. The prototype may not be competitive with popular uni-processor software yet, but should return correct results for input data sets of any reasonable size. The purpose of this research is to demonstrate effective methods of parallelizing a maximum parsimony branch & bound algorithm for phylogenetic inference. This will allow researchers with knowledge of possible optimizations in parsimony searches to combine them with the
framework developed in this thesis, hopefully providing great performance improvements over all uni-processor versions in the future.

2.3 Possible Uses

In addition to providing a prototype for future, more sophisticated and market competitive parallel software for phylogenetics, this thesis will hopefully serve other purposes as well. Dr. Buckley [7], of the Biological Sciences department here at RIT originally introduced me to this area of computing, and to the need for a parallel parsimony program. Besides providing research into the area of parallel computing for phylogenetic inference, the program would serve an equally useful purpose with his students, as they use a uni-processor version of the software already. Aside from the actual program written to implement the methods of parallelization used for this thesis, this document will provide a useful explanation of the inner workings of the implementations of parsimony algorithms in phylogenetics. Finding documented research of this type proved to be very difficult, and this thesis could be of use to anyone looking for specific details of phylogenetic inference software development. On a final note concerning the usefulness of this research – a scalable, cluster-independent parallel implementation provides the opportunity for performance increases limited only by the magnitudes of the clusters used which, given the growth rate of the internet today, are increasing rapidly. As with any parallelized software, the
outlook for gaining faster results through access to more powerful resources in the future, are very good; the opposite is true for uni-processor versions, which are bound by the physical properties of a single microchip and its highest associated processing capabilities. With computing resources such as IBM’s Blue Gene [11] being developed, and the proliferation of open-source based Beowulf clusters using networks of workstation (NOWs), the prospects for growth in parallel computing are favorable.
Chapter 3 — Maximum Parsimony

Most phylogenetic algorithms for optimal evolutionary tree searching can be broken down into two parts:

- The first includes algorithms responsible for keeping track of which tree topologies have been searched and for finding new ones to evaluate.

- The second involves the methods used to rank particular trees in order to determine which of them are more likely than the others to be the optimal trees (given the search criteria)[4].

Using algorithms in graph theory, a field of mathematics, it can be guaranteed that every possible tree is considered in the search; the second part of searching for evolutionary trees though, is much more ambiguous. A multitude of methods have been proposed to rank trees according to the probability that they explain the evolution of the given data set accurately [4]; however, none of them can be proven to be completely right or wrong all the time[4]. Maximum parsimony is a method of ranking, or scoring bifurcating trees and will be used in this thesis. The definition of parsimony is: “economy in the use of means to an end” [24]. In essence, parsimony assumes that the most likely path of evolution will be the one requiring the fewest number of changes. This chapter will examine this algorithm in closer detail.
3.1 The Data

DNA (Deoxyribonucleic Acid) is present in all living organisms and can sometimes be related to the physical traits of an organism[23]. It is therefore a good candidate for use as input data to phylogenetic inference algorithms. Although the implementation in this thesis will be based on the use of nuclear or chromosomal DNA data, other cellular structures such as mitochondria also possess DNA (mtDNA) which can be used as well, and the parsimony algorithm described below is the same for other morphological characteristics, as it is for DNA. Obviously, searching using the entire DNA of a human would be infeasible, as this would include more than three billion bases, or characters. Instead, a smaller segment of the entire DNA (no more than 5000 in the case of this thesis) which are chosen by a biologist based in part on the level of differentiation of the target DNA sequence among the species of interest, is used as the input data for a particular species. Combining these same portions of DNA taken from multiple species, forms the data used in phylogenetic inference. Composed of the four different nitrogenous bases adenine (A), cytosine (C), guanine (G), and thymine (T), the final input data set is an M-by-N matrix of M characters (A,C,G,T), for each of the N species represented. Figure 3.1 below summarizes what a sample 8-by-5 sized data file's data could look like for a few made-up species.
<table>
<thead>
<tr>
<th>Species</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human</td>
<td>C C A G T C C T</td>
</tr>
<tr>
<td>Chimpanzee</td>
<td>A C A C T A A G</td>
</tr>
<tr>
<td>Fly</td>
<td>C C G T C G C T</td>
</tr>
<tr>
<td>Cat</td>
<td>A C A T G T A G</td>
</tr>
<tr>
<td>Lion</td>
<td>G C A T A T G A</td>
</tr>
</tbody>
</table>

Figure 3.1: Sample input data for a typical phylogenetic program.

3.2 The Algorithm

Within the category of phylogenetic analysis using parsimony, multiple methods have been established to implement the search. Before getting into the details of the Fitch parsimony algorithms used in this thesis though, other algorithms and their differences will be discussed.

3.2.1 Weighted vs. Unweighted

The concept of weighting certain changes in state of a particular character site in the DNA data is possible in some algorithms in order to address the issue that certain characters are more likely to change to certain others (i.e. C → T or G → A is likely, but C → G or T → A is unlikely) [6]. The state change probabilities are based on a combination of physical facts and assumptions made by biologists, and are usually summarized through use of an inverse probability or cost matrix, relating state changes as shown below.
Figure 3.2: Sample cost matrix for a weighted parsimony algorithm.

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>0</td>
<td>2.5</td>
<td>1</td>
<td>2.5</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>2.5</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2.5</td>
</tr>
<tr>
<td>T</td>
<td></td>
<td>2.5</td>
<td>1</td>
<td>2.5</td>
<td>0</td>
</tr>
</tbody>
</table>

The Sankoff algorithm implements this type of weighted parsimony [6], but also with an associated computational cost when compared with strictly non-weighted methods. A non-weighted algorithm such as Fitch parsimony is essentially the Sankoff method with a cost matrix consisting of all 1's except in the diagonal, which consists of 0's (as it costs nothing to stay in the same state). The Fitch algorithm may not be as flexible as Sankoff parsimony, but for the purposes of this thesis it will be used, since the focus is primarily on parallelizing the search and not the scoring algorithm (which could be replaced by a more sophisticated one later). Also, un-weighted parsimony may not provide as many variables to customize a search for a particular application, but it is widely accepted, and still produces relatively accurate results [2].

3.2.2 Rooted vs. Unrooted

Closely related to the weighted or un-weighted factor is the issue of trees being rooted or un-rooted. Evolutionary trees are generally perceived as being rooted, since they begin with a root or common ancestor. A non-intuitive property, specific only to un-weighted searches though, is that groups of rooted trees can be represented by single un-rooted ones and still produce the same results [6]. This is because the parsimony
ranking algorithm counts the number of changes on a tree. Starting at any possible root on an un-weighted tree and counting changes across will result in the same score for ranking purposes. Given an un-rooted tree of \(n\)-taxa, there are \((2n - 3)\) rooted trees which can be formed by rooting the tree at various internal branches. Figure 3.3 below, shows a 4-taxa unrooted tree and the \((2 \times 4 - 5) = 5\) rooted trees that result from rooting the tree in each of the possible locations.

\[
\begin{align*}
&\text{A} \\
&\text{C} \\
&\text{T} \\
&\text{G}
\end{align*}
\]

\[
\begin{align*}
&\text{A} \\
&\text{C} \\
&\text{T} \\
&\text{G}
\end{align*}
\]

\[
\begin{align*}
&\text{A} \\
&\text{C} \\
&\text{T} \\
&\text{G}
\end{align*}
\]

\[
\begin{align*}
&\text{G} \\
&\text{A} \\
&\text{C} \\
&\text{T}
\end{align*}
\]

\[
\begin{align*}
&\text{G} \\
&\text{A} \\
&\text{C} \\
&\text{T}
\end{align*}
\]

\[
\begin{align*}
&\text{G} \\
&\text{A} \\
&\text{C} \\
&\text{T}
\end{align*}
\]

\[
\begin{align*}
&\text{G} \\
&\text{A} \\
&\text{C} \\
&\text{T}
\end{align*}
\]

Figure 3.3: A 4-taxa unrooted tree and the corresponding rooted trees it represents.
The computational benefit of using un-weighted trees is therefore two-fold:

- First, simpler and faster algorithms such as the one developed by Fitch can be used as opposed to more complex ones implementing the cost matrix requirements.

- Second, each un-rooted tree that is evaluated accounts for each of the rooted trees that could be formed by rooting it.

In section 2.2, it was shown that the number of possible bifurcating rooted trees for \( n \) taxa was found by the formula in figure 2.2. The number of un-rooted \( n \) taxa trees is therefore that number divided by \( (2n - 3) \), or equivalently, the same as the number of \( (2n - 1) \) taxa rooted trees. The equations in figure 3.4 show this:

\[
\text{trees}_{\text{unrooted}} = \frac{1 \times 3 \times 5 \times 7 \times \cdots \times (2n - 3)}{(2n - 3)}
\]

\[
\text{trees}_{\text{unrooted}} = 1 \times 3 \times 5 \times 7 \times \cdots \times (2n - 5)
\]

\[
\text{trees}_{\text{unrooted}} = \prod_{i=3}^{2n} (2n - 5)
\]

or

\[
\text{trees}_{\text{unrooted}} = \left[ \frac{(2(n-1)-3)!}{2^{(n-1)-1}((n-1)-1)!} \right]
\]
The number of unrooted bifurcating trees is given by the following formula:

$$\text{trees}_{\text{unrooted}} = \left[ \frac{(2n-5)!}{2^{n-2}(n-2)!} \right]$$

Figure 3.4: Number of n-taxon un-rooted, bifurcating trees that exist [6].

3.2.3 The Fitch Parsimony Algorithm

As mentioned, Fitch parsimony is the method used in this thesis for ranking un-rooted bifurcating trees. In addition to offering the benefits of using un-rooted trees since it is an un-weighted algorithm, it is also simpler to understand than most other methods of scoring trees. The algorithm proceeds as follows [8]:

Given an un-rooted tree topology specifying the leaf locations of the species:

1. For each column in the input data matrix
   
   a. Place the characters in that column from each species in the leaf locations on the un-rooted tree that they correspond to.

   b. For each group of 2 child nodes (starting at the leaves)

      i. If the child node characters (or character sets) are the same or have characters common between them, then keep their intersection and set the parent node as those kept characters. (Keep \(\{\text{leftnode}\} \cap \{\text{right node}\}\) if result of \(\{\text{leftnode}\} \cap \{\text{rightnode}\}\) is not the null set.)

      ii. If the child nodes have no common characters, add 1 to the score of the tree, and keep the combination of the two groups' characters. (Keep \(\{\text{leftnode}\} \cup \{\text{rightnode}\}\) and add 1 to the tree's score if result of \(\{\text{leftnode}\} \cap \{\text{rightnode}\}\) is the null set.)

20
Figure 3.5 below may clarify the above algorithm.

It is an example of Fitch parsimony for a 5-taxa tree using the characters from the first column of data in the sample input file data given in figure 3.1 above. Combining the first 2 leaf nodes \( \{A\} \cap \{C\} = \emptyset \), both sets have no common characters, and therefore 1 is added to the score for the tree (an asterisk next to a node's characters denotes that its formation caused an increase in the score for the particular tree). The same is the case for the last two leaf nodes \( \{A\} \cap \{G\} = \emptyset \), and also for the intersection of \( \{AG\} \cap \{C\} = \emptyset \). However, when we try to combine the sets \( \{AC\} \) from the left half of the tree with the \( \{ACG\} \) set on the right \( \{AC\} \cap \{ACG\} = \{AC\} \), there are common characters. The formation of the character set for this final root node therefore does not incur an increase in the score; however the evaluation of this first
column of input data did cause a score increase of 3, as can be seen in the diagram. This process is repeated for each column of the input data matrix, using the same tree. After all columns have been scored, the total number of nodes incurring score increases becomes the score used in ranking that un-rooted bifurcating tree. All possible un-rooted bifurcating trees must then be iterated through and accounted for, and the tree(s) containing the least number of nodes incurring score increases (asterisk nodes), is (are) the solution(s) to the parsimony search for optimal phylogenetic trees.

3.3 Pre-Scoring Optimizations

The algorithm described above uses Fitch parsimony to evaluate each and every column of the input data matrix against the tree being used. Typically though, in practical data that is derived from actual species which are often closely related, there are many patterns which can be exploited to avoid having to evaluate every column [6]. The most obvious of these is of course that if a column consists of only 1 character type A, C, G, or T, there will be no changes in the tree (or intersections that form the null set), and the score for that column will be 0. Similarly, if a column is composed of all one character except for 1 other character (e.g. AAAAG), the score for this column will be 1. Also, if a column has only 1 occurrence of 2 or 3 other characters (AAACG or AATCG) and the rest of the column is composed entirely of one other character type, the scores would be 2 and 3, respectively, for these columns. These four cases constitute what biologists call uninformative sites in the DNA, since the scores for
these columns will remain the same regardless of the tree that is being scored. Another obvious reduction in computation can be taken advantage of if one row is identical to another, in which case their scores will be the same. And finally, a less straight-forward reduction can be used when one row has the same pattern as another, in which case their scores will also be the same. For example, the columns AACGTT and GGTACC both have the form WWXYZZ where $W = A$ and $G$, $X = C$ and $T$, $Y = G$ and $A$, and $Z = T$ and $C$, for the respective columns.

Applying these ideas to the input data in figure 3.6 (taken from figure 3.1), we know that the first column's score is 3. The 2\textsuperscript{nd} column has no changes and will be 0. The 3\textsuperscript{rd} column contains one character that is different from the others in the column, which itself occurs once, and the score is therefore 1. The 4\textsuperscript{th} has two characters which are different than the rest of the column, each only occurring once, so the score is 2. Similarly, the 5\textsuperscript{th} and 6\textsuperscript{th} columns have three characters which are different than the rest of the row (although the rest of the column for 5-taxa is only two characters), each only occurring once, and the scores are both 3. Finally, column seven is the same as column one (score of 3), and column eight is the same pattern as column one (XYXYZ) for a score of 3 as well. Therefore the total score for the particular tree is 18. The algorithm would then move on to another tree and score it in the same way, keeping track of the tree topologies with the lowest scores to be reported as solutions after all possible bifurcating un-weighted trees have been accounted for.
<table>
<thead>
<tr>
<th>Species</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human</td>
<td>C C A G T C C T</td>
</tr>
<tr>
<td>Chimpanzee</td>
<td>A C A C T A A G</td>
</tr>
<tr>
<td>Fly</td>
<td>C C G T C G C T</td>
</tr>
<tr>
<td>Cat</td>
<td>A C A T G T A G</td>
</tr>
<tr>
<td>Lion</td>
<td>G C A T A T G A</td>
</tr>
</tbody>
</table>

Score: $3+0+1+2+3+3+3+3 = 18$

Figure 3.6: Scored input data matrix for the tree topology in figure 3.5.
Chapter 4—The Branch and Bound Shortcut

Although Fitch parsimony is one of the quicker scoring algorithms due to its simplicity, the nature of the search space growth given an increase in the number of input taxa, increases far too quickly for this method of searching when using data sets of 12 taxa (given a reasonably sized computer cluster) or larger [4]. Although sometimes biologists must resort to non-deterministic heuristic searches, which estimate the optimal tree, there is still another very important optimization which can reduce search times considerably for deterministic searches like Fitch parsimony. The Branch and Bound strategy [10] works by “pruning” the search space, eliminating entire groups of trees. The algorithms starts by first scoring trees with fewer taxa than there are in the input data, and then adding one taxon at a time, re-scoring the tree each time. If at any point the score of a tree containing a reduced number of taxa is already higher than the best score found for a full-size tree (with as many taxa as are represented in the input file) so far in the overall search, we know already that adding additional taxa could only worsen the score, and we can omit scoring any trees that can be built from this reduced-taxon tree which we found a higher score for.
4.1 Tree Topology Representation

4.1.1 Tree Identifiers

Before the Branch and Bound algorithm can be implemented, we must develop a way of keeping track of which trees have been searched, which ones need to be, and which ones can be eliminated. Ultimately, we need a method of associating each individual tree topology for any number of taxa, to a certain unique identifier. At first glance, simply numbering the trees would seem a viable option; however, as we saw in section 2.2, the number of trees would quickly overflow the range of even an unsigned 64-bit integer. The range of such a variable would be $1.84467 \times 10^{19}$, and with a 20-taxa input data matrix, there are $2.21543 \times 10^{20}$ possible un-rooted trees.

A more robust way of identifying trees uses a variable sized vector of numbers, and is based on the principles used to determine the number of possible trees for a given number of taxa presented in section 2.2. As discussed earlier, there is only one 3-taxon un-rooted tree, then three 4-taxon trees which can be formed from the 3-taxon one, five 5-taxon trees from any of the three 4-taxon trees, and so on. From here on, I will refer to a reduced-taxon tree as a tree with fewer taxa than are present in the input data file (fewer than the optimal solution trees which will be found). Also, the term taxa-level will be used to identify the number of taxa in the reduced-taxon tree (for example, a 6-taxon reduced-taxon tree reduced by one taxa-level would contain 5-taxon). If we use a vector to keep track of which reduced-taxon trees were used at each taxa-level to
achieve the current tree, then we will be able to uniquely identify any possible tree with any number of taxa. Therefore we will have an identifier vector of numbers in the range 1→1, 1→3, 1→5, 1→7, ... for the 3rd, 4th, 5th, and 6th, ... taxa-levels respectively. This can be easily represented using an array of numbers and leaving the first three numbers blank. Doing this allows indexing corresponding to each taxa-level. See figure 4.1 below for a visual representation of this.

Figure 4.1: Tree identifier vector example.
Figure 4.1 above shows the 3rd, 4th, and 5th taxa-levels, and the associated identifier vector values for each of the trees (only the middle treeID[4] = 1 tree is expanded to save space).

4.1.2 Tree Generation and the Newick Format

Since we now have a system of identifying trees, we must also be able to convert a given identifier into a symbolic form representing a tree's topology, so that it can be stored and manipulated on a computer. An open source program called TreeGen [28] does exactly that, and was adapted in this thesis to accept the type of tree identifier vectors discussed above instead of being limited to one 32-bit integer identifiers (which it originally used). After some modifications, the portion of the software that was used now accepts a vector identifier and returns a Newick formatted string corresponding to the un-rooted tree identifier. The Newick format is a shorthand notation for representing a tree symbolically using parenthesis and commas to represent branching [4]. It is fairly intuitive to understand; figure 4.2 provides a couple of examples of tree topologies and their corresponding Newick string representations.
Having now outlined a way of identifying all possible tree topologies for any number of taxa, as well as having a method of generating and storing a tree topology symbolically on a computer, we have the necessary tools to use the Branch and Bound algorithm. As mentioned earlier, starting with the single 3-taxa tree, taxa are added one at a time in every possible way. From now on, we will refer to the score for the topology found with the fewest required changes so far, or lowest score, as the threshold score. The threshold score and identifiers for full trees using all taxon from the input file are kept track of by the program. If a taxa named $E$ is added, and the score for that reduced-taxa tree becomes greater than the threshold so far, the algorithm undoes the addition of taxa $E$, returning to the reduced-taxa tree that was added to, and tries to add the same taxa $E$ but in a different way. Each time the algorithm backtracks in this way, it effectively skips evaluation and ranking of all trees.
that could have been built using that tree which produced a score higher than the threshold. If the algorithm has tried to add the new taxon $E$ in all possible ways and still produces a score greater than the threshold, the taxon $D$ that was added before the new taxon $E$ trying to be added, is also removed. This process of returning to a lower taxa-level could continue until the single 3-taxon tree is left over, in which case all trees have been accounted for. The algorithm is outlines below (based on [6, 4]):

Given the 0th 4-taxon un-rooted tree as a starting point:

1. While the algorithm hasn't backtracked to the single 3-taxon tree
   a. Score current tree
   b. If (tree score $\leq$ threshold)
      i. If (taxa level = # of input file taxa)
         1. save tree, replacing other best trees and their score so far if this tree had a better score.
         2. Remove the last added taxon and add it in a different way, or return to the next lowest taxa-level if that taxon has been added in all possible ways (continue to lower and lower taxa-levels if the taxa added at those level have also been added in all ways, but stop at taxa-level 3)
      ii. Else
         1. Add another taxon to the current tree.
   c. Else
      i. Remove the last added taxon and add it in a different way, or return to the next lowest taxa-level if that taxon has been added in all possible ways. (continue to lower and lower taxa-levels if the taxa added at those level have also been added in all ways, but stop at taxa-level 3)
In the above algorithm, it is not the actual trees which are being manipulated, added to, reduced, and so on; rather it is the identifier vectors described earlier that are being added to. For example the first of the 5-taxa trees in figure 4.1 has identification indices as follows: [C]1[1]/[C], which correspond to the ways in which the 3rd, 4th, and 5th taxa were added. In the algorithm above, if we were to remove and add the last added taxon in a different way (step 1.b.i.2 or 1.c.i.), we would simply increment the 5th taxa-level index to obtain [C][1]/[C]. If in these steps (1.b.i.2 or 1.c.i.) we reach [C][1]/[C] and cannot add the 5th taxon in a different way, we return to the lower taxa-level and add that 4th taxon in a different way to obtain [C]/[2] (note that the 3rd index, or 5th taxa-level index is unimportant). If we need to add another taxon as in step 1.b.ii.1, and we were at the 4th taxa-level using the above [C]/[2] tree, the new 5-taxa tree identifier would be [C]/[3]/[0] (note that the 5th taxa-level index, or 3rd index is reset to 0 when a tree identifier is stepped to the next higher taxa-level). Finally, to tie everything together, the visual representation shown below in figure 4.3 may be of further assistance to understanding the algorithm. Starting with the single 3-taxa tree A1, the score is obviously lower than the threshold (which should be set to a very high number at first) and taxon D is added to obtain B1, which is also lower than the threshold. The C1.1 group of trees is continued until all trees have been searched in the same manner; however, trees C1.2 and C1.3 at the 5th taxa-level were skipped since their scores must have been higher than the threshold set from one of the trees built from the C1.1 tree. After the C1.5 group of trees is searched, the E taxon cannot be added in any other
way, and we return to B1, then add D to the C branch instead of the B branch obtaining B2. The B2 tree at the 4th taxa-level must be lower than the threshold; however the C2.3, C2.4, and C2.5 trees were not. Finally, after returning from the B2 group of trees, we add D to the A branch obtaining B3, which is already higher than the threshold. Now out of ways to add the D taxon at the 4th taxa-level, we return to the 3rd taxa-level in which there are obviously no other ways to add the C taxon. This means the search is complete.
Figure 4.3: Illustration of Branch and Bound [4].
Chapter 5 – Parallelization of the Search

The Branch and Bound algorithm is not unique to phylogenetic inference; it can be effectively employed to improve performance in any other field involving similar types of searches for trees. As such, parallelization of the algorithm has also been the topic of other research projects[12], papers[1], and literature[27]. As with any parallel algorithm, there is a need to be able to divide the problem into smaller ones, while keeping track of what has been done and what else needs to be accomplished to finish the job. The tree topology identification system outlined in section 4.1.1 was designed with parallelization in mind. Instead of sending each of the individual tree topologies that need to be ranked to the various nodes in the cluster, sending the information describing a single reduced-taxa tree topology in the form of an identifier vector, can implicitly represent all of the possible trees that can be formed by adding taxa to it. This will allow for both the partitioning of the problem, as well as a means to track progress and avoid missing any tree topologies or inadvertently processing other ones multiple times.

5.1 Characterization of the Problem

The parallelized code is written using the freely available MPICH libraries in C, and will be tested on message passing clusters. These clusters will include the 16-node Linux cluster in RIT's Computer Engineering department (segfault.ce.rit.edu), a 30-
node NOW (Network of Workstations) comprised of the AMD Opteron VLSI machines also from the Computer Engineering department, and possibly the 104-node cluster in the Golisano College of Applied Computing. All of these clusters will also be homogeneous, meaning each of the nodes' hardware and software will be identical. When parallelizing any algorithm on non-shared memory systems such as those described above, there are a number of common goals for the results and performance of the parallel code which tend to recur. These include:

- Reducing the communication-to-computation ratio. Ideally, if a program parallelizes in a perfectly linear fashion (speedup is equal to the number of nodes used), this ratio would be 0. The added overhead of sending data to be processed to the nodes, and then gathering their results because memory isn't shared, should be minimized as much as possible.

- Reducing the amount of idle time on nodes during execution. Aside from waiting for communication to finish before further processing, ensuring that no node becomes unproductive simply because it runs out of data to be processed is also a major concern.

The branch and bound problem imposes a few other concerns as well. As mentioned before, sending tree identifier vectors implicitly provides tree topology information to a receiving node for all of the trees that could be built from it. Unfortunately, it is not usually possible to use a taxa-level which provides the same number of reduced-taxon...
tree identifiers as there are nodes, since the number of trees for a given taxa-level is \((2n - 5)\). Furthermore, even if 1, 3, 15, 105, ... nodes are used to split up the reduced-taxa trees for the 3rd, 4th, 5th, 6th, ... taxa-levels, it is highly unlikely that they will all evaluate the same number of trees and complete their work in the same amount of time. By going to higher and higher taxa-levels, the number of reduced-taxa tree identifiers increases quickly, and a more even distribution of these tree identifiers occurs. Of course, as more tree identifiers are being sent across the network, the communication-to-computation ratio suffers.

Central to the effectiveness of branch and bound, is finding a tree topology which generates a very low score, thus eliminating a majority of the other possible trees with scores higher than this value. Finding a new threshold that is lower than the previous one by even 1 or 2 changes in DNA can have immediate effects on the algorithm's ability to reduce the search space. It is therefore imperative that in a parallel search, nodes finding improvements in the threshold value are able to share them with all other nodes, and that there is a mechanism in place for this. Ideally, thresholds would all instantly update when one node finds an improved threshold value. As is the case with sending too many reduced-taxa tree identifier vectors though, sending too many threshold updates across the networks would also have unwanted effects on the communication-to-computation ratio.
5.2 Previous Work

One of the only references I found for this thesis that focused primarily on the parallelization of branch and bound algorithms was [1]. Unfortunately, the portion which dealt with the flavor of branch & bound used for phylogenetics, was based on the use of a shared memory system. Previous to my attempt at parallelizing this problem, another graduate student (Ken Jacobi [12]) from the Computer Science Department at RIT worked with Dr. Buckley [2] as well, to parallelize the same branch and bound parsimony search. Although he was unable to get a version of his software working which provided correct results, his work and ideas for parallelization were a good starting point for this thesis. His research provides a good overview of the basics of phylogenetic analysis, parsimony, and the branch and bound algorithm.

5.3 Solution Considerations

After taking the above-mentioned goals into consideration, a number of combinations seem possible for the parallelization given the trade-offs between sending more or less data across the network, the communication-to-computation ratio, and the performance of the program. The choices which must be made regarding which parallelization scheme to use can be divided into the three categories that follow.
53.1 Tree Topology Distribution Strategy

Two distinct strategies were considered, both with their own advantages and disadvantages. The first simply sends groups of trees out to each node including the root (through reduced-taxa tree identifiers), collects all of their results once each of them has completed, and repeat the process until all possible tree topologies have been accounted for. The advantage of this barrier-type parallelization method is that all nodes including the root are used; the disadvantage however, is that since some nodes will process less trees than others (because of the branch and bound algorithm), there will almost always be some idle nodes while others are finishing, before new tree identifiers are sent out. The second strategy is to delegate the root node as the scheduler. Instead of searching its own group of trees as in the first strategy, it would wait for results from other nodes as soon as they complete, and immediately return them a new group of trees to work on. Here, the advantage is that none of the working nodes will go idle. The drawback is that we must sacrifice the root node's processing capabilities by making it wait for replies from others.

53.2 Grain Size

For both of the above strategies we have ignored the fact that the number of reduced-taxa tree identifiers may not be divisible by the number of nodes being used. As said before though, the time taken for nodes to evaluate a given group of tree topologies represented by a tree identifier can vary dramatically due to a number of criteria.
Therefore even if we can represent the entire search space by fifteen tree identifiers and we send them to fifteen working nodes, some nodes will go idle if they finish early. By sending smaller and smaller sized search spaces to each node, thus decreasing the grain size, the nodes may be given new work to do many times. This would solve the problem of having a majority of the nodes idle while a small number of them finish. The drawback is that there is more communication needed, increasing the communication-to-computation ratio. Tests must be conducted to determine the optimal grain size as it affects the inverse relationship between node idle time and the communication-to-computation ratio. Furthermore, to obtain the same grain size (or group of tree topologies represented by a given tree identifier), tree identifiers of different taxa-levels must be used for the various numbers of possible taxa in the input file.

5.3.3 Threshold Updating

The last of the three variables to consider in optimizing the parallelization of branch and bound parsimony, concerns keeping all nodes concurrently aware of the best threshold found by any node so far. There is no question that this feature needs to be incorporated somehow. The ambiguity lies in determining how often updates should be made. Having all nodes report and update their thresholds frequently ensures that they will all benefit from the latest threshold, which could allow for greater reductions
in search space. Making the threshold update intervals too close though, will flood the network and increase the communication-to-computation ratio.

Ultimately, the decision for which combination of the above three items will be most effective has to be determined through testing and analysis. In Chapter 7, the overall results of the program as well as results for specific tests targeted towards finding the best combination for parallelization, are presented.
Chapter 6 – Implementation

So far, only theory on the development of the program in this thesis has been discussed. This chapter will focus on the actual implementation it, and on the reasoning for each of its components. As mentioned before, the C programming language was used, and parallelization was accomplished through the MPICH libraries (a free implementation of MPI [16]). Also, a GUI (graphical user interface) written in QT [5, 22], was added as a wrapper to the main functionality of the program (which can still be accessed through command line control).

6.1 Module Summary

Each of the specific modules of the program will now be discussed in further detail; to begin though, a summary of the purposes and functions of each of the files that make up the program is given below, as these files will be referenced throughout the module descriptions.

- **treeID.(c/h)** Simplifies the use of tree topology identifier vectors by providing functions to support the creation of these vector, as well as other necessary operators for this new data type.

- **init.(c/h)** This module is used at the beginning of a search. It checks the validity of the specified input DNA file and populates a data structure, which is
used throughout the program using information taken from that file. Various pre-search optimizations (discussed in section 3.3) are attempted as well, to reduce the search time.

- **treegen(c/h)** Given a unique tree identifier vector, the corresponding tree topology in the Newick string representation is returned. Part of the implementation for this module was borrowed from [28].

- **score(c/h)** Using the Fitch parsimony algorithm described in section 3.2.3, the DNA data matrix from the input, and a tree topology given in Newick format, this module will return the corresponding score for the data on the particular topology.

- **worker(c/h)** This module is given a single reduced-taxa tree identifier vector and oversees the search through all possible topologies that could be built from it. It implements the branch and bound algorithm; using the DNA input data, it iterates, and if necessary scores each of the possible tree topologies in its search space while also keeping track of the lowest scoring ones.

- **multi_proc(c/h)** Oversees all other modules. It initializes other nodes, distributes the total search space using reduced-taxa tree identifiers, then collects and reports results.
- **tree_graph(c/h)** Converts a Newick string tree topology to an ASCII-based picture of that tree (a dendrogram). This was not a simple task; it was therefore implemented as its own module, which takes a Newick string as input and draws the corresponding dendrogram on the screen.

The flowchart below shows how these modules work together collectively.
Figure 6.1: Flowchart overview of program execution and used modules.

In the above diagram, the modules are indicated by the labeled black boxes surrounding them, the calls from one computer or module to another are shown by arrows (with description text nearby), and the life of the program on the root and slave nodes is shown by the vertical dashed lines (only one slave node is shown; however
the flowcharts for any of the others would be the same, since the root node interacts with them in the same ways).

Finally, an efficient and simple way of containing and communicating many different forms of data related to the search was needed. For this, a new structured datatype was created, which contained information such as the DNA data, pre-scoring information, best trees found so far, program performance statistics, and more. This new type called \( DNA\_info \) allowed all information that could be needed by any part of the program to be passed from module to module through a single pointer to a memory address.

6.2 Input Data Format

Many of the popular phylogenetic inference software suites allow robust input data formats; however, given the goals of this thesis, implementing most of these features seemed out of scope. The input format was kept very simple (but is still compatible with many phylogenetic programs such as PHYLIP [7]), and is composed of two non-whitespace character strings per line. The following rules outline the data format:

1. Each taxon must be represented by a single line in the file.

2. Each taxon line must contain exactly 2 non-whitespace tokens, separated, preceded, and/or followed by any amount of whitespace.
3. The first token must be used as the identifier name for that particular taxon, and the second must be used as the input DNA data.

4. The second token containing the DNA data must be composed of only the following characters: A,C,G,T. All DNA data tokens (2\textsuperscript{nd}) must contain the same number of characters.

5. Any blank lines will be ignored. Any lines with one, or more than two non-whitespace tokens will be flagged as a bad format error, and the program will terminate.

Figure 6.2 below, provides a sample 5-taxon input file, and an alternate form that is not as aesthetic, but is valid and equivalent to the first.

<table>
<thead>
<tr>
<th>beginning file 1</th>
<th>beginning file 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human AGTCCTAGC</td>
<td>Human AGTCCTAGC</td>
</tr>
<tr>
<td>Chimpanzee GCCATCGCT</td>
<td>Chimpanzee GCCATCGCT</td>
</tr>
<tr>
<td>Fly GCTGCCAAG</td>
<td>Fly GCTGCCAAG</td>
</tr>
<tr>
<td>Cat ACTGACAAA</td>
<td>Cat ACTGACAAA</td>
</tr>
<tr>
<td>Lion GATACATGG</td>
<td>Lion GATACATGG</td>
</tr>
<tr>
<td>end of file</td>
<td>end of file</td>
</tr>
</tbody>
</table>

Figure 6.2: Two valid and equivalent 5-taxon input data files.
6.3 Initialization and Pre-Score Optimization

Upon running the program, the *init* routine (described earlier) is called. If the file opens properly and is of valid format, the pre-scoring optimizations (section 3.2.3) such as finding uninformative sites, are done. For each site (or column) in the data, there are essentially three possibilities for the way it should be scored:

1. The site is uninformative, and the contribution to the overall score can be immediately calculated to be 0, 1, 2, or 3.

2. The site is informative, and no other site so far (in the process of examining the columns from left to right) has either been identical or had the same pattern. In this case the site should be marked for later evaluation using Fitch parsimony for the particular tree topology being evaluated.

3. The site is either the same or has the same pattern as a site to the left of it that has already been examined. In this case the site should be marked with a reference to the site it is equivalent to, so that the score contribution from that site can be used for this particular site as well (once the other site's is known).

The information above is saved using the *scores*, and *referal* arrays contained in the *DNA_info* structure datatype. Other essential information is also populated in the structure within the *init* routine, such as *num_sites*, and *num_taxes*, as well as the actual matrix of DNA character stored in the *num_sites-by-num_taxes* sized matrix called *data*.
6.4 The score and treegen Modules

6.4.1 The score Module

The score module implements the Fitch scoring method and for a given DNA data matrix and tree topology, it uses the pre-scoring data to produce the total score for the tree using all DNA data. The module is initialized with the DNA and pre-scoring data when the root initializes the slave nodes. After this, whenever the worker module calls the score module, only a Newick string representing the topology is needed. Iterating through the sites in the DNA matrix, the score module keeps a total score. When needed, the Fitch algorithm is used, and the column's data is inserted into the tree topology. For example, if the column data for a 5-taxa tree was AGAGT (for the 0th through 4th taxon respectively), and the Newick tree topology string given was(((0,4),(2,(1,3))), the algorithm would start with this tree: (((A,T),(A,(G,G)))). Just as the algorithm prescribes (see section 3.2.3), the tree is then reduced to a single group of characters, one pair of leaf nodes (with a common parent) at a time. Figure 6.3 below shows the remainder of this procedure, where each asterisk represents 1 being added to the score for this site against the particular tree. The first group of taxa immediately separated by a comma is always reduced first, until there are no more commas in the Newick tree, and there is only one group of characters. This process always completes in (# taxa− 1) steps, 4 in this case.
Original tree: 

\[(A,T),(A,(G,G))\]

1st Reduction: 

\[(A,T),(A,(G,G))\] #

2nd Reduction: 

\[(A,T),(A,G)\]

3rd Reduction: 

\[(A,T),(A,G)\] #

4th Reduction: 

\[(A)\]

Figure 6.3: Example of Fitch parsimony scoring reduction for a given site in the `score` module.

The actual characters used to save the A,C,G, and T nucleotides in the software are actually 0x23 → 0x26. This leaves characters '0' through '~' (char 0x30 → 0x7E) so that Newick trees can be formed for up to 79 taxa. The `score` module is one part of the program which accounts for a large portion of processor utilization during execution, and would be a good candidate for replacement by a more efficient algorithm. An efficient parsimony implementation was not the goal of this thesis though; more ideas for possible improvements to the program's performance will be given in chapter 10.

6.4.2 The `treegen` Module

The implementation of the `treegen` module, as said before, was partially taken from an open source program called TreeGen, written by Marty J. Wolf of Mankato State University [28]. Since parallelization of the search was the focus of this thesis, and finding an efficient way to iterate through each possible topology did not appear to be a simple task, TreeGen code which does just that was adapted to work with the treeID topology identification vectors. Through a single method interface, any tree identification vector is converted into a Newick tree string using characters 0x30 → 0x7E to represent taxon # 0 through up to taxon # 78.
6.5 The treeID and worker Modules

6.5.1 The treeID Module

Key to the program's ability to evaluate more than 12 taxa (there are more 13-taxa un-rooted trees than the range of a 32-bit integer), is the tree identification vector type that was discussed in section 4.1.1. The treeID module provides a number of methods to facilitate the use of these vectors. The new method allocates space for a vector of given size corresponding to the number of taxa to be represented. The indID and dedID methods increment and decrement the last position in the vector for the number of taxa being used, and return a non-zero number (otherwise a 1) when there is a carry (indID) or borrow (dedID) to the lower taxa-levels in the vector. The cmpID returns 0 only when two vectors representing the same number of taxa are identical (represent the same topology). The appID method copies a vector to another of the same or greater size (representing the same number of taxa or more). Finally, the printID method can be used for debugging to display the contents of a vector. This utility was created to simplify the code throughout the program, especially in the worker module, which will be discussed next.
6.5.2 The worker Module

The worker module implements the branch and bound algorithm. It can be used on either the root node or slaves; however, for the methods chosen to parallelize the search (discussed in the next section), actual searching and scoring will only be accomplished on slave nodes. Upon initialization, it passes on the information contained in the DNA_info structure that it received to the score module, which uses it later. Once a slave node receives a reduced-taxa tree identifier vector from the root, it passes this vector and the best threshold so far to the worker module, and the search begins. The identifier vector given will also have an associated reduced-taxa level for the reduced-taxa tree it represents. The branch and bound search that follows will consider every possible tree that can be built by adding up to all the remaining taxa in the input to the reduced-taxa tree given. Following the algorithm for branch and bound discussed in section 4.2, the taxa-level is incremented and decremented according to whether or not the score for the current tree is higher or lower than the threshold. Branch and bound continues until the initial reduced-taxa tree identifier gets incremented, meaning all trees that could possibly be built from that first tree have either been evaluated and/or skipped. Whenever a tree with the number of taxa in the input is found with an equal or lower threshold than the best so far, it is saved. The DNA_info structure has fields named num_best_trees and best_trees, which as their names suggest, hold the number of best trees (if there are multiple topologies with the same score) found so far and their corresponding topologies stored as treeID identifier
vectors, respectively. The best_trees vector is doubled in size whenever it becomes too small to fill the number of best topologies found. Finally, the worker module is also responsible for giving periodic threshold updates to the root node, which then replies back with the lowest global threshold among all slave nodes. This will be discussed further in the next section, which covers the choices made for parallelization, and why those methods were picked over others based on the related data collected and show in chapter 7.

6.6 The multi_proc Module

Finally, the module which ties all others together and also contains the main method called from the command line is multi_proc. After successfully using the init module to read in the input file, do pre-scoring, and populate the DNA_info structure, it then initializes the slave nodes by sending them this structure. On the slave nodes, the multi_proc module initiates a blocking MPI_Recv call which waits for instructions from the root. If either there was an error or the input data contains less than 8 taxa (in which case the processing is all done on the root node instead of the slaves), the slaves are simply notified that they will not be needed and are instructed to terminate execution. If the slaves will be needed though, multi_proc will oversee the distribution of all reduced-taxon trees to them, and manage threshold updates throughout the search.
6.6.1 Parallelization Architecture

Along with the reduced-taxon trees representing the search spaces to be handled by the slave nodes, threshold values also need to be sent out for use with the branch and bound algorithm in the worker module. As mentioned before, the lower this value is, the more search space can be skipped. Therefore it would seem important to find a suitably low initial value for it before sending it out. There are many ways to do this. Short heuristic searches would seem to be a good option, but would also involve a significant amount of extra development. The chosen method for this thesis was to simply iterate through a number of randomly generated tree topologies (having the same number of taxa as the input file). Choosing the right number of these trees to search initially on the root node is difficult to do, as enough should be processed to find a decently low threshold; however, too many will waste the processing capabilities of the slaves which are idle and waiting for work. Arbitrarily, the number of random trees to search first in this thesis was set to 1 percent of the search space, but no more than 10000.

Once the initial value for the threshold has been set, and the slave nodes have been initialized and are waiting for input, the program is ready to begin searching. As mentioned before though, determining how to distribute the data is the crux of this thesis. The three main variables that were analyzed were: 1) the overall architecture of the parallelization, 2) the grain size, and 3) the threshold update interval.
Initial tests were done for various sized input files and varying numbers of cluster nodes to measure processor utilization, for both architecture schemes considered. From chapter 5, these schemes were: 1) to use the slaves and the root for processing and use a barrier to control when they were all sent the next set of reduced-taxon trees, or 2) to not use the root node for processing, but have it constantly ready to accept slave node results and send them new spaces to search. The initial tests mentioned above showed that there was a definite gain in overall processor utilization for the 2nd scheme (see section 7.1), and so it was used.

The second variable examined was grain size, or the taxa-level used for the reduced-taxon trees sent to the slave nodes. Decreasing grain size did have positive effects until a certain point, at which excess communication began to take its toll. Unfortunately, tests attempting to find these exact points varied depending on the nature of the specific input data used. Short of writing code to examine the data and attempt to calculate a suitable grain size, or fixing the grain size for any data, the only other reasonable simple implementation was to base the grain size on the number of taxa in the input file. Using the data from test results (see section 7.1), it was found that simply making sure there were at least twice as many reduced-taxon trees as there were processors, whatever tax-level was required for that, worked quite well in keeping all nodes evenly utilized throughout the search.
The last of the three variables, the threshold update interval, was analyzed in much the same way. Using the data from these tests, it was found that one second intervals worked as effectively as any faster update rate, without adding the communication load that they would. This update interval can be changed before compilation in the `multi_proc.h` file. Given the above choices for parallelization, the resulting algorithm followed the pseudo-code shown in figure 6.4 below:

6.6.2 Algorithm Implementation Pseudo-Code

Given: the taxa-level is set, the reduced-taxa treeID at that taxa-level is reset to the first tree, and there are trees to be processed

1. WHILE there are reduced-taxa trees left to be processed OR there are busy nodes
   a. IF there are nodes available (not busy).
      i. Send a free node a reduced-taxa tree and threshold.
      ii. Mark this node as busy.
      iii. Increment the treeID to get the next reduced-taxa tree, and if all reduced-taxa trees at this level have been iterated through mark the "trees_left" flag as false.
   b. IF there is something to receive from a slave node.
      i. IF it is results.
         1. Compare results' threshold with best global threshold for results received from all nodes so far, and save the node's best trees if the threshold is better or equal to the global one.
         2. Mark this processor as not busy or idle.
      ii. IF it is a threshold update.
         1. Compare the received threshold to the best global threshold received so far, and replace it with this new one if there is an improvement.
2. Send the root node's best global threshold to the slave node the update was received from.

2. Notify all slave processors to terminate, as all reduced-taxa trees have been searched.

Figure 6.4: multi proc module parallel search algorithm pseudo-code.

In section 1.b of the algorithm in figure 6.4 above, the slave node always sends a number first, which is its current best threshold. If this number is negative, it signals the root that the slave simply needs an update, and this number is then negated again to obtain the slave's current threshold. If the number is positive, this indicates that the slave node has finished its work, and has results. After the algorithm executes step 2, the search has finished. Results are then displayed, and the program exits.

6.7 Output Format and Implementation of the GUI

6.7.1 The Output Format

The format of the output can be specified when executing the program from the command line through switches, or from the GUI through checkboxes. The dendrogram checkbox in the GUI, or the -d switch for command line execution, displays a visual representation of the branch topology of the tree along with the taxon names. The newick option (-n) displays the corresponding Newick trees with the taxon names inserted. Either, both, or neither of these two options can be specified. The Newick
format, which uses commas and parentheses, is compatible with many tree graphing utilities that are available as well (such as those available with PHYLIP).

6.7.2 The GUI

Finally, a GUI was implemented for the program. Instruction for compiling and using the GUI, as well as the rest of the program, will be discussed in the appendix. The Qt toolkit from Trolltech [5,22] was chosen as the interface API for this task. Not only was it free, but it also has support for cross-platform compatibility. The GUI is essentially a wrapper program which performs an operating system call to the multi_pntext program. It uses the exec system call to run multi_pntext with the specified number of nodes. Once complete, it opens and parses the output file to obtain results information for display in the GUI. Progress information for the current search when one is running is displayed in the terminal window the GUI was started in.
Chapter 7—Results

This chapter summarizes the results that were found and recorded for a number of different tests of performance and functionality. First, charts will be provided to support the decisions made in the previous chapter regarding the methods used to parallelize the search as efficiently as possible. The methods used to verify the validity of the results obtained by the search will then be discussed. The performance of the program will be shown through speedup graphs for various input files. Finally, a comparison of performance on the sfgait cluster versus the clustered network of workstations in the computer engineering VLSI lab at RIT will be presented. As mentioned previously, other factors such as the characteristics of the input data, can play a significant role in the complexity of the search. Therefore the optimizations that follow in this chapter are simply attempts to optimize the program for the input data that was chosen. Through this, the effects of varying different aspects of the parallel program will hopefully be demonstrated.

7.1 Parallelization Architecture Optimization

In the previous chapter, decisions were made regarding the manner in which the search would be parallelized. As was mentioned, these decisions were based on experimentation and analysis. The three criteria were: 1) Parallelization architecture -- the barrier method, or using the root solely to distribute reduced-taxa trees when
needed, 2) Grain size – The taxa-level of the reduced-taxa trees to be sent to the slave nodes, and 3) the time intervals between threshold updates for the slave nodes.

At first glance, finding a way to effectively optimize these three variables seemed difficult, since it was thought that making changes to one of them would have significant effects on the others. Testing and analysis of all possible combinations would be a difficult task. After some thought and experimentation, it was discovered that certain of the three variables seemed to have more influence over the others. Of the two considered options for variable 1, the barrier approach seemed to be intuitively less efficient than simply providing a slave node with more work as soon as it finished.

![Figure 7.1: Distribution strategy option 1 -- Barrier.](image)
its previous search (the 2nd option for the first of the 3 parallelization criteria). The barrier method would take as long for each set of search spaces sent out, as the time taken by the slave node that is slowest in completing its search. Figure 7.1 illustrates this first option, where all nodes in the cluster complete their work before any of them start new searches. The second option would allow nodes that have finished early to continue processing, and only go idle near the completion of the entire search, when there are no more reduced-taxa trees to be sent out. This is shown in figure 7.2. Since

![Distribution strategy option 2 - dynamic.](image)

Figure 7.2: Distribution strategy option 2 – dynamic.

uses the root node solely for distribution and coordination of the search though, the efficiency of the program as a whole is immediately reduced to:
\[ 100\% - \left( \frac{1}{\# \text{ Of } \text{ Nodes}} \right) \]

**Figure 7.3:** Loss of efficiency when the root node does no searching.

Fortunately, the efficiency improves as the number of nodes used increases, and will approach the efficiency of using all nodes for large enough clusters. Focusing scalability in this way, this architecture option again seemed favorable over the other. Finally, as the grain size is reduced, and each slave node processes several separate reduced-taxa trees (or search spaces), the problem of nodes going idle while waiting for the end of the entire search (when no more reduced-taxa trees are available), becomes negligible. This is similar to the loss in productivity due to the root node not searching, and improves when the number of nodes used is increased. The chart below shows the average percentage of time nodes spent searching, where the first architecture is the barrier scheme, and the second involves using the root solely for distribution of reduced-taxa trees. The measurements for the second scheme take into account that the utilization of the root node is 0%.
It was assumed that using a taxa-level of 7 would still maintain a large enough grain size to not affect the communication-to-computation ratio significantly. This will be verified when analyzing variable 2 (grain size), next. As the chart shows, the results were as expected, and the 2\textsuperscript{nd} architecture is favorable for higher numbers of nodes. The second variable of interest was tested using the 2\textsuperscript{nd} architecture scheme, assuming that it would be the choice for variable 1. The chart below summarizes the results for varying the taxa-level for the 2\textsuperscript{nd} architecture scheme. The test was performed using 6, 11, and 16 nodes (using an 11-taxa input file) to ensure that results would not be adversely affected by scaling the number of nodes. The chart clearly shows that taxa-levels 6 and 7 are favorable, and similar results were found for the same test using the 1\textsuperscript{st} architecture scheme.
The last of the three variables, the threshold update interval, did not have as significant an affect as the first two; it does however provide an increase in performance when properly optimized. After analyzing the first two variables, it was clear that the 2nd parallelization scheme was favorable, and so it was used for the analysis of the 3rd variable. The 11-taxa input file was used again, as it provided a large enough search space to make the execution time long enough that changes in the variables of interest have a measurable effect, but is also short enough to prevent having to run tests that last multiple hours or days. The chart below shows the results from the test which showed that one second was a favorable interval to use.
7.2 Functionality Verification

The results were of course proven to be correct before any performance analysis was attempted. Before continuing with a discussion of the tests performed to verify the correctness of the results, it may be helpful to see exactly what the output from the program looks like. Without using the GUI, the textual version of the program outputs a detailed report of the findings. As an example the following command was run:

```
mpirun -np 16 multi_proc testing_dat/t9.dat -n -d
```
The call to `mpirun` must be made first with the number of processors (-np 16, 16 nodes in this case) to use 2nd, and then the MPI program to run, `multi_proc`. The first argument to the program must always be the input DNA file, `t9.dat` in this case, which looks like this:

<table>
<thead>
<tr>
<th>Species</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>chipmunk</td>
<td>GTCGCGGATAAGCGGCGATGCCTTC</td>
</tr>
<tr>
<td>squirrel</td>
<td>GTCACAGAAGAGCCGAGCTGACCT</td>
</tr>
<tr>
<td>cat</td>
<td>CGCGCGCGCTCGCGCCTCTTGACT</td>
</tr>
<tr>
<td>dog</td>
<td>CTCAACAAACTCAAGCTGACCTTT</td>
</tr>
<tr>
<td>cow</td>
<td>CAGTCAAGCAGTACGTCAGCTCAAACCT</td>
</tr>
<tr>
<td>chicken</td>
<td>TGAAATCCAATCAGCTACGAGCATGACG</td>
</tr>
<tr>
<td>snake</td>
<td>ACTAGCAGCACTGACGACGATGACTCC</td>
</tr>
<tr>
<td>horse</td>
<td>GCTACGATCGACTACGTCGCTGACAA</td>
</tr>
<tr>
<td>butterfly</td>
<td>GACTGACTGACTGACTGACTGACTGACG</td>
</tr>
</tbody>
</table>

Figure 7.7: Sample Input Data.

Running this program with the -d and -n options will output the resulting optimal tree(s) in both Newick string format and the dendrogram (visual) format, respectively. The following is the actual output of this program:
TOTAL TIME = 1.544549 seconds
INPUT FILE = "testing_dat/t9.dat"
NUM TAXA = 9
NUM CHAR = 28
NUM PROCS USED = 16
BEST SCORE = 112
TREES FOUND = 1
TREES SEARCHED = 51.06 %
AVG. PROC UTILIZATION = 98.58 %

Tree #0
(0, ((3, (4, (2, (5, 8))), (1, (6, 7)))))

chipmunk

| dog
| cow
| cat
| chicken
| butterfly
| squirrel
| snake
| horse

Figure 7.8: Output data corresponding to the input file in figure 7.7.

The results show execution time (search time only), information about the input file, the best score, the number of trees with that score, the percentage of all trees that could not be skipped by the branch and bound algorithm, and the average utilization of all slave nodes that did processing. Finally, the Newick and dendrogram formats of the optimal trees are displayed (note: optimal trees are numbered starting at 0).

This output was then compared with the output of a program that was known to be correct (PAUP was used), using a branch and bound parsimony search for the exact
same file. Multiple (3) tests were done using all possible numbers of nodes (1 through 16), for a randomly created input file of 4, 5, 6, 7, 8, 9, 10, and 11-taxa. Selected tests were done for input files with higher numbers of taxa and with real DNA data from Dr. Buckley, since these tests took much longer to complete. The largest file tested was a 20-taxa file, which took most of a day to complete; however the program should be able to process more (up to 78, limited by the scorer module). For each test it was verified that the best score, number of trees with that score, and their corresponding topologies were the same as those generated by PAUP. It should be noted that PAUP uses consensus trees wherever possible, and the program in this thesis does not. Consensus trees use multifurcating trees to represent multiple bifurcating trees. For example the three trees: (A, ((B, C),D)), (A, (B, (C,D))), and (A, ((B, D),C)), in this program would all be represented by a single tree (A, (B, C,D)) in the output of a PAUP search.

7.3 Speedup Results

As with any parallel program, one of the best indicators of its scalability is a speedup graph. Ideally, using 16 nodes would allow a program to have a speedup of 16, or run 16 times faster. Tests were run multiple times and averaged, for both the 9-taxa and 11-taxa randomly generated input files. Figures 7.9 and 7.10 below show these results.
Figure 7.9: Speedup for 9-taxa.

The choice to use the root node for only the distribution of work and not for performing actual searches, is readily apparent in these graphs. Using one node, no communication is necessary, as all work is done on the root node. When using two though, since the root is always delegated to not perform any searches when multiple nodes are in use, the added communication necessary to relay all search information to the single slave node actually increases the execution time from what it was when searching on the root. As discussed earlier though, this scheme is more scalable than the barrier-type design considered, where the root is used for searching as well.
7.4 Performance on Other Clusters

Verifying that the program performed well on other platforms as well as on the one used thus far for development (the sqfadt cluster), was necessary to prove the functionality and usefulness of this project. Tests for performance, data integrity (correctness of results), and basic functionality were performed on these other platforms. At the onset of this thesis, it was hoped that there would be an opportunity to test the program on the 104-node Bioinformatics and College of Applied Computing cluster. Unfortunately, when the program was ready for testing, the cluster
was disabled so that it could be rebuilt. The VLSI computer lab workstations were on the list of testing platforms to be used as well though, and after clustering them using MPICH, this 30-node NOW (network of workstations) still allowed for a reasonably good test of scalability.

![VLSI cluster 11-taxa speedup](image)

**Figure 7.11:** Speedup for 11-taxa on the 30-node cluster of VLSI workstations.

The AMD Opteron processors on these workstations were superior to the Celeron processors of the 16-node `sgefault` cluster, and are responsible for the improved execution times (node-for-node) shown in the graph below.
Figure 7.12: Execution time for the same input on VLSI workstations vs. the segfault cluster.

On average, the VLSI computers were 1.51 times faster for the results collected, than the computers in the segfault cluster. (Multiple trials were done on each cluster for each input file tested, always using 16-nodes).
Chapter 8 – Difficulties Encountered

This chapter will outline some of the difficulties I came across in this thesis which I found to be the most challenging. Gaining an understanding of several areas of study, as well as developing a system of identifying an unlimited number of tree topologies were two of the major ones.

8.1 Multi-Disciplinary Research

Research literature related to this thesis was quite sparse, especially for computer implementations of algorithms such as maximum parsimony. In addition to the algorithms only being described in basic terms, there were multiple algorithms which all achieved the same goal of searching through parsimony, some of them faster or slower, and simpler or more versatile. It took roughly a month and a half of reading bioinformatics literature to understand the terminologies used to describe the parsimony algorithms. It was only at this point that I was able to begin implementing the program, after choosing to use the Fitch parsimony algorithm. Equally difficult was understanding the branch and bound, and parallelization algorithms. For example, if parallelization techniques were described in a paper (such as [1]), the descriptions were typically very detailed, but targeted towards platforms other than the types I would be working with, such as shared data parallel computers. The program implemented in this thesis drew upon information from a number of different fields;
Bioinformatics for the parsimony and branch and bound algorithms, graph theory in mathematics for an understanding of tree generation, and computer science and engineering for the parallelization of the search. The time spent investigating each of these areas was one of the major cruxes of the project. The areas I understood least, such as Bioinformatics, accounted for much of the lower level implementation which had to be done first and understood well, before parallelization could be attempted.

8.2 The Tree Topology Identification Problems

The toughest problem to overcome was undoubtedly that of finding a way to uniquely identify all trees being considered for searches when using more than 12 taxa. The tree generation code developed by [28] had not been designed to work for more than 12 taxa, and thus simply used integers for topology identifiers. After some thought about the problem, I had hoped to be able to use a different data type, such as a long long integer, which was a 64-bit integer on the machines being used. Again though, this solution was limited, and the number of trees that could be represented would be insufficient for searches with more than 20 taxa. The solution that was ultimately chosen, involving the identifier vectors described in previous chapters, was discovered by examining the Treegen code used to translate its integer identifiers to corresponding topologies. The numeric identifier was first made into a vector before the topology was created. Given a 6-taxa integer tree identifier of 103 for example, the Treegen algorithm finds the equivalent vector representation by successive modulus and
division operations using the number of ways branches can be added at each successive taxa-level, starting with the last index in the vector. For this example, 103 would be divided by the number possible branch additions at each taxa-level. This may sound confusing, but it is quite simple. The possible taxa-levels are 6, 5, 4, and 3 (for unrooted trees) for this 6-taxa tree, and the identifier will have 4 indices corresponding to the branches taken at each of these levels. The number of possible branches to take (ways in which a single taxon can be added) at a given taxa-level is \((2^n - 5)\), giving 7, 5, 3, and 1 for the taxa-levels. 103 \( \mod \) 7 = 5, we now know that the index for the last place (taxa-level 6) in the identifier vector is 5. 103 / 7 = 14, and 14 \( \mod \) 5 = 4, and the index for the 5th taxa-level is 4. Continuing this we find 14 / 5 = 2 and 2 \( \mod \) 3 = 2, 2 / 3 = 0, and 0 \( \mod \) 1 = 0 (the 3rd taxa-level index is always 0 since there is only 1 3-taxa tree). We are left with the vector [0][2][4][5], which is equivalent to the 103 integer. Figure 8.1 below summarizes the steps that were taken:

Given the 6-taxa integer identifier = 103

103 \( \mod \) (2(6) - 5) = 5 and 103 / 7 = 14
14 \( \mod \) (2(5) - 5) = 4 and 14 / 5 = 2
2 \( \mod \) (2(4) - 5) = 2 and 2 / 3 = 0
0 \( \mod \) (2(3) - 5) = 0

The vector is the remainders [0][2][4][5] = 103 (integer identifier)

Figure 8.1: Conversion of an integer to a vector identifier used in TreeGen [28].
Ultimately, this entire step was removed, as tree identifiers were stored as vectors in the new version of the program’s method of tree identification. To accommodate these changes though, a group of support functions including operators (treeD.c and treeD.h) were added, since the branch and bound algorithm must manipulate tree identifiers extensively to keep track of where it is in the search. Currently, the identifier consist of unsigned char elements, which have a range of 0 to 255. The index limits in the vector are 0→ 0, 0→ 2, 0→ 4, etc… for the 3rd, 4th, 5th, … taxa-levels. Also the 0th, 1st, and 2nd positions in the array that implements the identifier vector are unused to reduce human error when referring to taxa-levels (i.e. to access 6th taxa level in the array called tID, simply use tID[6]). By using the tree identifier vectors, the new limit on the number of taxa that can be used in the input is \((255 \div 2) = 127\), minus 3 for the first indices which aren’t used, allowing for 124-taxa trees to be represented (although the score module currently limits this to 78-taxa trees).
Chapter 9—Conclusion

Quite early in the research, it became evident that producing a commercial grade application for branch and bound parsimony, that rivaled or improved upon the performance of well known software such as PAUP or PHYLIP, would not be feasible given the timeframe of this thesis. Years of research by many people have been invested to optimize uni-processor searches; gaining an understanding of all of these methods and implementing them would be a very large task even for someone previously familiar with bioinformatics. As stated in the proposal, the main focus of my research was to find possible methods of parallelization for branch and bound parsimony algorithms. Having a speedup of 12 for 16 nodes, as the results in section 7.3 show, the prototype search developed met this goal adequately. The current implementation of the program parallelizes the problem in a manner that is scalable and efficient. (The shortcomings in execution time for this program versus PAUP and other widely available parsimony programs are primarily related to the highly-optimized scoring algorithms that are implemented in these programs, as this tends to be the major bottleneck in phylogenetic search software). Through experiments conducted in this thesis though, many of the nuances associated with parallelization of branch and bound parsimony search problems were discovered and analyzed. The effects of varying grain size, the methods of workload distribution and tracking of the search, as well as the effects of varying the threshold update intervals were all
considered. As mentioned, all of these variables affect one another, and therefore need to be evaluated in a somewhat iterative process. Central to the variances in performance that result from changes to these three variables, was the communication-to-computation ratio, as more communication could often distribute the workload better while also incurring added communication time which could have had a negative effect on execution time. The importance of these three variables and their relationship with the parallelization (see section 7.1) of branch and bound parsimony algorithms could benefit others looking to parallelize similar searches. The values found to be optimal for the program in this thesis were optimized for the safiut cluster and the chosen input data; however, on a different cluster with different hardware components and input data, these values would need to be re-calculated in order for the program to perform optimally.

9.1 Effects of Input File Data on Parallelization

Another implication that was discovered, which could affect the values chosen for the three variables, is the nature of the data in the input file and how well that data is suited to the branch and bound algorithm. For example, consider the two following data files:
The first file has random data, and contains more changes than most files containing real DNA data to be analyzed. The second contains data in which each taxa contains exactly one 1 change from the one listed before it, which also would not be likely to occur in nature. The branch and bound algorithm can eliminate almost all of the search in the second input file though, and only needs to search less than 0.001 % of the entire search space, as opposed to the 10.81 % required for the 1st file. The search using the file with 17-taxa takes about 40% less time to run than the one containing
11-taxon, simply because the branch and bound algorithm can take advantage of the data. This is an unrealistic and extreme case; however, although it does not need to be considered for uni-processor algorithms, it can have a significant effect on which values are chosen for the three aforementioned parallelization variables. It shows that if a perfectly optimized parallel search were to be accomplished, the data in the input would need to be analyzed first.
Chapter 10 – Future Possibilities

As mentioned before, this program is not yet comparable in performance with commercial software of its kind. Although it does contain valuable information for any developer who may be parallelizing a phylogenetic search, there are a number of improvements that could significantly enhance the program's performance and/or facilitate its use. This chapter will summarize a few of the ideas that I had for possible features to add to the program, but did not have time to implement. Of the six topics that will be discussed, the first three will involve improvements to the parallelization of the search, and the last three are not related to parallel searching, but may improve the program as a whole.

10.1 Threading on the Root Node

The two architectures for parallelization described throughout this thesis were the barrier scheme, and a second type in which the root node did no processing, but simply delegated work to slave nodes. The reason the root node cannot do work in the second scheme as it must wait (or poll) for incoming messages from slave nodes. The solution to this would be to make the root process multi-threaded. Thus, once the program began searching, the root node could spawn a separate 2nd thread to do processing, and share CPU time with the first thread which would still be responsible for responding to messages from slave nodes. Since the root node would have two
tasks, its performance would likely be lower than that of slave nodes; however, it is likely that the CPU is idle most of the time using the current non-threaded scheme, and the new, threaded version would be able to use this idle time more effectively.

10.2 Dynamic Grain Size

Currently, the grain size is determined through tests and is then fixed for a given compilation of the code. As mentioned in the conclusion chapter, input file data can have a significant effect on runtime. Although the tests that were run optimized the parallelization variables for the chosen input data, it is impossible to setup the program to run optimally for any input data using the current scheme. To improve upon this, the root could change the taxa-level it is using for current reduced-taxa trees which it is sending to the slaves depending upon how long the slaves took to process the previous trees they were assigned. In this way the program could optimize itself as it ran. For large enough data sets, this would be more efficient than fixing the parallelization variables as they are now.

10.3 Pre-Search Input File Analysis

The two input files that were compared in the conclusion chapter brought to attention the need to consider the type of data contained in the input, if optimal performance is to be achieved. A module to analyze the input data, determine how well suited it will be for the branch and bound algorithm, and then set the parallelization variables
accordingly could dynamically maintain performance from one input file to another. Perfecting this type of module would be difficult, and a better understanding of the branch and bound algorithm would be needed; however, even a basic prediction based on observations made from the input DNA data, could help the overall performance of the program.

10.4 Improved Initial Threshold Determination

The search for an initial threshold value to send slave nodes used in the program for this thesis is very crude. It is random, and could even search topologies that have already been evaluated. There are numerous options for better initial threshold searches. For example, the program could start with a few reduced-taxa trees and add taxa one at a time, in the ways that incur the least addition to the tree's overall score, to achieve trees with as many taxa as are contained in the input file (keeping the lowest scoring one). An even better solution would be to use a heuristic search for a short period of time and keep its best result.

10.5 Faster Fitch Parsimony

The biggest bottleneck in the program is the scoring algorithm, as this is the heart of the search, where the CPU spends the most time. This is the case with any parsimony search, and for this reason scientists and developers have gone to great lengths to improve the efficiency of scoring. The researchers in [9], and [18] offer detailed insight
into some of these speedups, and go as far as implementing a Fitch scoring algorithm in assembly pseudo-code to save clock cycles. Again though, optimization of the parsimony algorithm was not the main goal of my research.

10.6 Other

Finally, there are a number of smaller changes that could be made to make the program easier to use, and more comparable in its input and output specifications to PAUP or PHYLIP. Consensus trees, as mentioned in chapter 9, could be used to summarize the output data through a more usable form. Also, the input characters currently accepted as valid are A, C, G, and T. The program would be more flexible if it accepted summarizing characters such as Y, which represents the group of nucleotides ACT; or even better, if it accepted the same .nex files used with PAUP as an alternative form of input.
Appendix A

This extra section is a user's manual for anyone wishing to use the program or code. It is divided into three sections: requirements and installation, use of the program, and limitations.

A.1 Requirements and Installation

To install and run at least the textual version of the program developed in this thesis you will need:

- At least one computer (or multiple networked computers).

- A C compiler.

- MPI libraries and modules for compilation and execution.

To use the GUI, you will also need:

- Access to the Qt libraries [5,22] during compilation

- A windows environment (Xwin server was used with Linux).

Assuming the reader understands how to install and use the above items, once the code from the accompanying CD is downloaded to the target platform, you will need
to compile it. To compile the main part of the program (textual version), run the following command from the directory the code is in:

mpicc score.c treegen.c tree_graph.c treelD.c worker.c multi_proc.c -o multi_proc

This will produce the executable named multi_proc. To compile the GUI program (which calls the textual program), run the following two commands to produce the executable ThesisGUI:

moc -o moc_ThesisGUI.cpp ThesisGUI.h

g++ -I$QTDIR/include ThesisGUI.cpp thesisGUItest.cpp
   moc_ThesisGUI.cpp -L$QTDIR/lib -lqt -o ThesisGUI

A.2 Using the Program

Once compiled, the textual program can be run by using the mpirun call as follows:

mpirun -np <num_procs> multi_proc <input_filename> <other options ... >

In the command shown above, the number of nodes to use must be specified in the <num_procs> space, and the input file in the <input_filename> space immediately following the program name (multi_proc). The following options can be specified (in any order) after the input file has been specified:

- <output_file> – If specified, all output will be directed to the specified file.
• -n – If specified, the Newick string format representations of the optimal tree topologies found will be included in the output.

• -d – If specified, the dendrogram format representations of the optimal tree topologies found will be included in the output.

A number of different error messages can be produced if a problem is encountered when starting the program. The list below describes what each of them will probably mean if they are encountered:

• *There was an error opening file: <input file>* – This could indicate a number of different problems that can occur while trying to open the input file (before parsing); more than likely, the specified file does not exist.

• *You must specify at least an input file* – The program needs an input file name token after the MPI call to run the program. You may have only given the -n or -d options, or no program arguments at all. The first argument found for the program will be used as the input file name.

• *Invalid Option: <option>* – Only the -d and -n options are currently valid. Trying to use the -m option for example, would result in this message.

• *The input file is not of the proper format* – This could mean a number of things related to the input file being formatted incorrectly including but not
limited to: More or less than 2 tokens found on a line, or invalid characters used for DNA data.

- **One or more of the taxa do not have the same number of characters** -- All taxa must specify DNA strings of the same length or number of nucleotides.

- **Illegal number of taxa in file. Program takes 4 \( \rightarrow \) MAX_TAXA only.** -- A four-taxon file is the smallest file that will be searched, since there is only one three-taxon un-rooted tree. The most taxa the program will accept in an input file is specified by MAX_TAXA, in *multi_procb*. 


The GUI program will look like figure A.1 below when it is first run:

![GUI interface diagram](image)

**Figure A.1:** The GUI when it is first opened.

The input and results (output) files and their paths can be easily specified on the lines provided. **When using the GUI, the results file must be specified for the program to operate properly.** The Newick and dendrogram formatted trees can be included in the output by simply checking the boxes under the “Output Tree Format” title. Also, the number of nodes to use can be specified using the up/down box titled “Number of Processors to Use”. Once the “Start Search” button has been clicked, the
"Status" will change to a green color and say "Running". If there was an error running the program, the messages will be shown in the terminal window in which the GUI was run from. During the search, the progress will be available in that same terminal window as well. Finally, when the search has completed, the results will be displayed in the GUI, and the "Status" will return to "Stopped", the GUI will then look something like figure A.2 below.

![GUI Example](image)

**Figure A.2:** An example of the GUI after it has finished a search.
A.3 Limitations and Special Settings

The program currently has the following limitations:

- Maximum Number of Taxa: 78. Although searching more than 20 taxa (the highest number tested) would likely take more time than is worthwhile with the current implementation, a theoretical limit of 78 is imposed by the scoring algorithm.

- Maximum Number of Characters per line in the input file (taxa name, spaces, and DNA characters): 4096. Can be changed in \texttt{init.h} by editing \texttt{MAX\_LINE\_LEN}.

- Only A, C, G, and T are accepted as valid characters for nucleotide data (see section 6.2 for more input file format specifications).

Of the three variables affecting parallelization of the search that were discussed throughout this thesis, two can be easily modified if the user wishes to experiment with them.

- Threshold Update Interval. Can be changed by modifying \texttt{THRESHOLD\_UPDATE\_INTERVAL} in \texttt{multi\_prch}.
Grain Size. The grain size is set just before the main while loop in the `distribute_tras0` method of `multi_proc`. The grain size is set by modifying the taxa-level, stored in the variable called `taxa_level`, and is not currently changed throughout the rest of the search.

The third of the three parallelization variables is inherent to the structure of the code written and would require re-writing the `distribute_tras0` method in the `multi_proc` module.
Bibliography


http://geta.life.uiuc.edu/~gary/programs/fastDNAml.html


[19] Schurr, Theodore G. “Mitochondrial DNA and the Peopling of the New World”.


http://paup.csit.fsu.edu/Paup_Doc_31.pdf


http://www.answersingenesis.org/creation/v20/i4/evolution.asp