Adapting the Phylogenetic Program FITCH for Distributed Processing

Robert A. Dubin

Follow this and additional works at: https://digitalcommons.montclair.edu/etd

Part of the Computer Sciences Commons
Abstract

Adapting the Phylogenetic Program FITCH for Distributed Processing
By: Robert A. Dubin
Thesis Advisor: Dr. Dajin Wang

The ability to reconstruct optimal phylogenies (evolutionary trees) based on objective criteria impacts directly on our understanding the relationships among organisms, including human evolution, as well as the spread of infectious disease. Numerous tree construction methods have been implemented for execution on single processors, however inferring large phylogenies using computationally intense algorithms can be beyond the practical capacity of a single processor. Distributed and parallel processing provides a means for overcoming this hurdle. FITCH is a freely available, single-processor implementation of a distance-based, tree-building algorithm commonly used by the biological community. Through an alternating least squares approach to branch length optimization and tree comparison, FITCH iteratively builds up evolutionary trees through species addition and branch rearrangement. To extend the utility of this program, I describe the design, implementation, and performance of mpiFITCH, a parallel processing version of FITCH developed using the Message Passing Interface for message exchange. Balanced load distribution required the conversion of tree generation from recursive linked list traversal to iterative, array-based traversal. Execution of mpiFITCH on a Beowulf cluster running 64 processors revealed maximum performance enhancement of up to ~28 fold with an efficiency of ~ 40%.
Adapting The Phylogenetic Program FITCH For Distributed Processing
by
Robert A. Dubin
A Master’s Thesis Submitted to the Faculty of
Montclair State University
In Partial Fulfillment of the Requirements
For the Degree of
Master of Science
March 2007

School: College of Science and Mathematics
Department: Computer Science

Thesis Committee:
Thesis Sponsor: Dr. Dajin Wang
Committee Member: Dr. Roman Zaritsky
Committee Member: Dr. Chunguang Du

Date: 3/26/07
Dean: Dr. Robert Rezant
Department Chair: Dr. Dorothy Deemer
ADAPTING THE PHYLOGENETIC PROGRAM FITCH
FOR DISTRIBUTED PROCESSING

by
ROBERT A. DUBIN

A THESIS
Submitted in partial fulfillment of the requirements
For the degree of Master of Science in
The Department of Computer Science in
The Graduate Program of
Montclair State University
March 2007
Acknowledgements
I am particularly grateful to my Thesis Advisor, Dr. Dajin Wang, for the support and advice he provided during the course of this study. In addition, I wish to thank Dr. Roman Zaritski for generously granting me access to his Beowulf clusters and for helpful discussions concerning their use, and Dr. Chunguang Du for suggesting this thesis project. Lastly, I thank David for his unconditional support throughout the years.
# Table of Contents

Abstract i  
Acknowledgements ii  
Table of Contents iii  
List of Figures and Tables iv  
Introduction 1  
FITCH: A Distance-based Heuristic Using Least Squares Criterion 2  
  Sequence Comparison and Distance 2  
  The Least Squares Criterion 3  
  Heuristic Approach Using Sequential Addition and Rearrangement 4  
  Task 1: Sequential Addition 4  
  Task 2: Rearrangement 4  
    Local Rearrangement 5  
    Global Rearrangement 5  
  Fractional Effort per Task 5  
Distributed Design 6  
  Two Design Options 6  
  Distributed Design of mpiFITCH 8  
Converting Recursive to Iterative Tree Generation 9  
  Recursive Tree Generation Is a Poor Model for Balanced Distribution 9  
  Array-Based Tree Generation Is a Superior Model for Balanced Distribution 9  
  Effects Of iterative_addtraverse() and iterative_rearrange() In FITCH Background 12  
Distributed Implementation 12  
  Dividing Up the Workload: Self-Assignment Based On Processor Rank 12  
  Iterative Tree Generation 13  
  Message Transmission Using MPI 13  
    Point-To-Point Message Passing: MPI_Send() and MPI_Receive() 14  
    Collective Communication: MPI_Allreduce() 14  
Update 15  
Performance 15  
Conclusions 18  
References 19  
Addendum I: Fitch Program Flow 22  
Addendum II: Fitch Trees and Their Growth 23  
  Abstract Tree 23  
  Data Structures of a FITCH Tree 23  
  Memory Allocation: A Branchless and Leafless FITCH Tree 24  
  The Initial, Three-Species Phylogenetic Tree 24  
  Simplified Representation of a FITCH Tree 26  
  FITCH Tree Growth by Sequential Addition 26  
  Adjacent Branch Swapping by Local Rearrangement 26
List of Figures and Tables

List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.</td>
<td>Distance-based branch length determination.</td>
<td>3</td>
</tr>
<tr>
<td>Figure 2.</td>
<td>Sequential addition.</td>
<td>4</td>
</tr>
<tr>
<td>Figure 3.</td>
<td>Local and global rearrangement.</td>
<td>5</td>
</tr>
<tr>
<td>Figure 4.</td>
<td>Brief pseudocode representation of FITCH algorithm.</td>
<td>6</td>
</tr>
<tr>
<td>Figure 5.</td>
<td>Two master-slave design options.</td>
<td>7</td>
</tr>
<tr>
<td>Figure 6.</td>
<td>Iterative, array-based traversal across a FITCH tree.</td>
<td>10</td>
</tr>
<tr>
<td>Figure 7.</td>
<td>mpiFITCH performance on clusters.</td>
<td>16</td>
</tr>
<tr>
<td>Figure 8.</td>
<td>Performance of mpiFITCH variants.</td>
<td>17</td>
</tr>
<tr>
<td>Figure 9.</td>
<td>Initialized tree in memory.</td>
<td>25</td>
</tr>
<tr>
<td>Figure 10.</td>
<td>The three-species FITCH tree</td>
<td>25</td>
</tr>
<tr>
<td>Figure 11.</td>
<td>Addition of fourth species to phylogenetic tree.</td>
<td>26</td>
</tr>
<tr>
<td>Figure 12.</td>
<td>Addition of fourth species at alternative positions.</td>
<td>26</td>
</tr>
<tr>
<td>Figure 13.</td>
<td>Adjacent branch swapping by local rearrangement.</td>
<td>27</td>
</tr>
</tbody>
</table>

List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1.</td>
<td>Recursive and iterative tree generation produce identical output on a single processor.</td>
<td>12</td>
</tr>
<tr>
<td>Table 2.</td>
<td>Load balancing among processors for a 50 species tree.</td>
<td>13</td>
</tr>
<tr>
<td>Table 3.</td>
<td>Execution of mpiFITCH on multiple processors has no effect on final output.</td>
<td>16</td>
</tr>
<tr>
<td>Table 4.</td>
<td>Speed up among distributed phylogenetic programs.</td>
<td>18</td>
</tr>
</tbody>
</table>
Introduction

Origin of species by common descent provides a framework that encompasses all aspects of modern biology. The evolution of two species from a common ancestor is believed to begin through the physical separation of two populations of a single species. Over time, accumulation of sufficient heritable differences through mutation warrants their classification into two separate, but related species. Paleontology and evolutionary biology attempt to deduce evolutionary relationships through physical examination of the fossil record. By contrast, phylogenetics attempts to identify and quantify evolutionary relationships through objective analysis of living species. These relationships are presented as phylogenies, more commonly known as evolutionary trees. A phylogenetic tree depicts a model of evolutionary descent based upon experimental data derived from living organisms that can include morphological traits, developmental studies, and/or gene and protein sequence similarities. The leaves (tips) of a phylogenetic tree represent living species (or in some cases, an extinct species); intermediate positions (internal nodes or other locations along the branches) represent species from the past that may or may not be known from the fossil record. The tree topology represents the pattern of evolutionary descent and branch lengths reflect a measure of evolutionary change. (Note that rate of evolutionary change is not necessarily coincident with the passage of time.) While the development of phylogenetic trees may in some cases seem esoteric, they are useful tools for studying human origins (Green et al., 2006), genetic diseases including cancer (Desper et al., 2000), disease susceptibility, and the origin, diversity, epidemiology, pathogenicity, and transmission of infectious diseases including tuberculosis (Mathema et al., 2006) and HIV/AIDS (Yusin et al., 2001).

For centuries, taxonomic classifications were developed through analysis of morphological features. While such groupings were useful, they did not necessarily represent descent from a common ancestor. With the development of sophisticated molecular biology tools such as protein sequencing, and in particular technologies that include gene cloning, polymerase chain reaction, and gene sequencing, great strides in our understanding of evolutionary relationships have been made through macromolecular analysis. This is especially true for analysis of the macromolecule responsible for common descent, the inherited genetic material DNA. Recent advances in gene sequencing speed and accuracy have resulted in the generation of massive amounts of sequence data from scores of different medically, agriculturally, and economically important genome sequencing projects, including biologically interesting model organisms (for example Aparicio et al., 2002). The accumulation of this data typically outpaces the rate with which it can be analyzed, mined, and exploited (Roos, 2001).

The three major tree-building methods are distance, parsimony, and likelihood (Hershkovitz and Leipe, 1998; Felsenstein, 1984; 2004). Distance methods use pairwise comparisons between two species to represent evolutionary change and attempt to identify tree topologies and branch lengths that best represent that data (Forey et al., 1992; Nei and Kumar, 2000). Parsimony methods attempt to generate a tree that requires the fewest assumptions (the least amount of evolutionary change) to best fit a data set. Likelihood identifies a tree with the greatest statistical probability of modeling data. Tree building approaches are algorithm or criterion-based. Algorithm-based approaches generate evolutionary trees through the iterative application of a list of rules. By applying an algorithm, a final, single tree is built up. Popular algorithms include the Unweighted Pair Group Method with Arithmetic Mean and the Neighbor Joining method, both of which are distance-based. By contrast, criterion-based approaches search the available universe of possible trees for a best tree. Searching attempts to examine all (or at least many) possible tree topologies, fitting the observed data to each topology and applying some criterion to select the
tree that best models the data. Popular criterion approaches include Maximum Parsimony and Maximum Likelihood.

Over the last 20 years, computer systems, including the Internet, have come to play an integral role in the organization, analysis, and presentation of biological data useful for gene discovery, drug discovery, and phylogenetic studies. A wide array of phylogenetic tree-building methods has been implemented for execution by computers (Hershkovitz and Leipe, 1998). This includes programs to align and compare sequences, generate phylogenetic trees, and view and interpret the results. The Phylogeny Inference Package (PHYLIP) is a freely available phylogenetic software package that is popular with the biological research community. Using DNA or protein sequences as input, PHYLIP programs identify and quantify evolutionary relationships among species and present the results as bifurcating, unrooted, evolutionary trees. Developed, written, modified, and maintained by Joseph Felsenstein (University of Washington) and his students, the PHYLIP web site (http://evolution.genetics.washington.edu/phylip.html) makes available for download the source code, executables, and documentation. This package contains programs that implement a variety of parsimony, distance and likelihood algorithms, and the programs run on a single processor.

Certain phylogeny-building algorithms are NP-hard problems (Day, 1986) and thus inferring phylogenies for large data sets can be computationally challenging. In the case of certain computationally intensive algorithms, particularly likelihood methods, the generation of large trees can be beyond the practical capacity of a single processor (Olsen et al., 1994). Distributed processing (Barney, 2007) provides a means for overcoming this bottleneck and the number of phylogenetic programs adapted for multiple processor environments is increasing (see references below).

The PHYLIP module FITCH builds phylogenetic trees through a criterion-based, distance method (Felsenstein, 1997) and is used extensively by plant biologists (Matsuoka et al., 2002; C. Du, personal communication). In an effort to accelerate research by expanding its utility, the goal of this research project is to adapt FITCH for parallel execution in a distributed environment. The resulting program, mpiFITCH, is a parallel version of FITCH (v. 3.56c) that executes on a cluster of separate, yet interconnected processors that communicate with one another over a network. The application programming interface selected to provide routines to support interprocessor communication is the Message Passing Interface (MPI). MPI was chosen as it is a “practical, portable, efficient,” and freely available industry standard for message passing (www.mpi-forum.org; Pacheco, 1997). Both FITCH and the supplementary source code required by mpiFITCH are written in C; importantly, the MPI-1 library is defined to extend the C programming language.

The remainder of this thesis describes an analysis of FITCH, and the design, implementation, and run-time characteristics of mpiFITCH when executed on two Beowulf clusters.

**FITCH: A Distance-based Heuristic Using Least Squares Criterion**

*Sequence Comparison and Distance*

Like all distance methods, the input data for FITCH is arranged as pairwise values in a distance table (Figure 1A). The experimental data used for generating distance tables previously included immunological cross reactivity and nucleic acid hybridization studies; today, however, more typical data sets include allele frequencies and gene / protein sequences. Each value in a distance table reflects some relative measure of evolutionary distance (divergence) between two species. For example, a simple method of converting gene sequence data into a distance value is to
determine the fraction of residues that differ for two aligned sequences (Figure 1B); this value could be refined further by assigning different weights to specific types of residue alterations (i.e.: transversion and transition; substitution and insertion-deletion). In this way, a single overall measure of difference is obtained for any two related strings; the fewer the differences, the more closely related the two homologous sequences, and by extension, the two species. (By contrast, parsimony and likelihood methods do not convert and discard the sequences; instead they use the sequences themselves as input and examine each residue for informative differences.)

**Figure 1 Distance-based branch length determination.** (A) Distance table for 4 taxa. (B) DNA sequence alignment for homologous genes from species A and B; basepair differences are indicated by *. (C) Equation to calculate sum of squares (Q). (D) Minimal branch lengths for the three, topologically distinct, 4-species trees determined by alternating least squares. Q values indicate degree of divergence from experimental data in (A) for each topology.

**The Least Squares Criterion**

While FITCH does not use the tree-building algorithm first introduced by its namesake (Fitch and Margoliash, 1967), it does retain the Fitch and Margoliash weighted sum of squares (Q) criterion (Figure 1C). FITCH uses Q in two ways, for branch length determination and for tree comparisons. FITCH builds up different tree topologies. For each tree topology, FITCH calculates optimal (minimal) branch lengths for a given data set by assuming that branch lengths are additive and by minimizing Q (Felsenstein, 1984; 2004). While FITCH could have determined optimal branch lengths by solving a set of linear equations, it instead makes use of a more rapid, alternating least squares method. As Felsenstein describes it, “when we make a small change of tree topology, the branch lengths of the resulting tree should change mostly in the regions that are altered, and rather little elsewhere. This means that the branch lengths from the previous tree provide us with good starting values for the branch lengths of the altered tree. [This] iterative algorithm for estimating branch lengths retains partial information at interior nodes of the tree. Thus, we not only retain the previous branch lengths, but we do not need to recompute the partial information at the interior nodes, at least not the first time they are used” (Felsenstein, 2004). This is approached by “…finding the least squares branch lengths for three branches of the tree at a time by ‘pruning’ all but those branches from the tree and solving exactly for the remaining three branch lengths. By repeating this operation for different parts of the tree, one approaches asymptotically a least squares solution” (Felsenstein, 1997). In addition to using Q to calculate optimal branch lengths for any given tree topology, it is also used to compare and rank trees of different topologies. For example, three topologically distinct, unrooted, bifurcating, 4-species trees can be drawn for the data set presented in Figure 1A. For each topology, FITCH calculates branch lengths that minimizes Q and identifies as best the tree with the smallest Q value (Figure 1D). The application of weighted sum of squares as criteria for both minimal branch length determination and tree comparisons makes this method quite attractive.
Heuristic Approach Using Sequential Addition and Rearrangement

With criterion-based tree-searching approaches, the theoretical goal is to examine all possible tree topologies and identify the best tree. However, since the number of topologically distinct trees increases rapidly with the size of the data set, analysis of all possible candidate trees is impractical when comparing more than ~10 taxa. For example, trees consisting of 5, 10, 15, and 20 species have 15, greater than 2,0000, greater than 7 trillion, and $2.2 \times 10^{20}$ topologically different, unrooted forms (Felsenstein, 2004). An inability to examine all possible topologies has spurred the development of heuristic approaches that attempt to identify highly plausible trees while not guarantying the identification of the very best tree (Sikes, 2007). The heuristic used by many PHYLP programs, including FITCH, is a combination of two tasks, “Sequential Addition” and “Rearrangement” (Felsenstein, 1995, 1997), also known as “stepwise addition” and “branch swapping,” respectively. These tasks are extremely similar and consist of repetitive execution of three ordered subtasks: the generation of a topologically new tree, the calculation of minimal branch lengths for the new tree, and the comparison of sum of square values between the new tree and the previously identified best tree. By analyzing significant portions of its source code and studying supplemental material available in the literature (Felsenstein, 1995; 1997; 2004) it was possible to develop a high-level description of the FITCH algorithm (Addendum I: Fitch Program Flow).

Task 1: Sequential Addition. With Sequential Addition, FITCH iteratively builds up its tree through the addition of species one at a time, with each iteration identifying a new, one species larger, best tree. Thus, FITCH adds the next (nth) species to the best (n-1) species tree at all $(2n - 5)$ possible positions, determines optimal branch lengths for each of the $(2n - 5)$ topologically distinct trees, and identifies the one tree with the lowest Q as the new, best n-species tree. The best tree identified by an iteration serves as starting point for the next iteration. The process begins by generating a unique, unrooted, three-species tree from the first three species in the distance table. The fourth species can be added to this tree at three possible positions, and a best four-species tree can be identified (Figure 2). In the next iteration, the fifth species is added to the best four-species tree at all five possible positions, and from these a best five-species tree is identified. While there are in fact 15 possible bifurcating, unrooted, five-species-containing trees, Sequential Addition generates and examines only five of them. In Fitch, Sequential Addition begins by a call to the function addtraverse(); the generation of all $(2n - 5)$ new trees occurs through recursive calls to addtraverse().

Figure 2. Sequential addition. The next (nth) species is added to the best (n-1) species tree at all possible $(2n - 5)$ positions. Minimal branch lengths are calculated and the tree with the lowest Q value is identified as the best n-species tree, which serves as the starting point for the next iteration. Not that all topologically distinct trees are not examined by this algorithm.

Task 2: Rearrangement. Trees generated exclusively by Sequential Addition are dependent upon the input data and the order that species are added to the growing tree. To reduce this order-of-addition artifact, Sequential Addition is supplemented with searches for better trees through
branch rearrangements. FITCH can perform two types of rearrangements. “Local Rearrangement” (also called nearest neighbor interchange) attempts to identify better trees by switching adjacent branches; “Global Rearrangement” (also called subtree pruning and regrafting) performs a more exhaustive search by removing subtrees and replacing them at all possible positions (Figure 3). While tree-building algorithms that combine Sequential Addition and Rearrangement significantly reduce input-order dependency, they still fail to guarantee that the very best final tree will be generated. For this reason, hundreds of different orderings of the input data are usually performed and analyzed (Felsenstein, 1997; Stewart et al., 2001).

Figure 3. Local and global rearrangement.
Local rearrangement swaps adjacent branches (i.e.: here C is adjacent to A, B, D, and E). An example of a global rearrangement shows subtree E/D being exchanged with branch A.

Local Rearrangement. In an attempt to identify an even better n-species tree, the best n-species tree identified by Sequential Addition is subjected to a series of Local Rearrangements that swap adjacent branches. For any n-species tree there are \((2n - 6)\) possible swaps that can be performed with adjacent branches. For each round of tree growth, Local Rearrangement generates, examines, and compares at least \((2n - 6)\) trees. For each new tree topology, branch lengths are minimized and sum of square values are compared. If none of the \((2n - 6)\) trees is found to be better than the best n-species tree that was identified by Sequential Addition, then Local Rearrangement terminates and the best n-species tree remains unchanged. By contrast, if an adjacent branch swap produces a better n-species tree, then that better tree immediately becomes the new best n-species tree, the current round of Local Rearrangement ends, and a new round of Local Rearrangement begins to examine a completely new set of \((2n - 6)\) trees. This type of algorithm is known as hill-climbing. Local Rearrangement initiates from a call to function rearrange(), which eventually calls addtraverse(); the generation of all \((2n - 6)\) trees occurs through recursive calls to rearrange() and/or addtraverse().

Global Rearrangement. Global Rearrangement is an optional feature in FITCH that is turned off by default. When the option is selected, it executes only following the addition of the last species. Global Rearrangement performs an extensive search for a better, final, tree through the removal and reinsertion of subtrees at all, not just adjacent, positions.

Fractional Effort per Task. Timed program runs of FITCH on a single processor personal computer (PC) and Linux box revealed that computational effort was divided unequally among the tasks. When FITCH was executed in the absence of Global Rearrangement, it was observed that Sequential Addition and Local Rearrangement accounted for 25% - 35% and 65% - 75% of runtime respectively, depending on final tree size (in this case, 50 and 75 species trees were examined). The inclusion of Global Rearrangement increased runtime by a factor of 4 - 5 fold and 13 - 18 fold when negative branch lengths (another option) were precluded and permitted, respectively. While I explored the possibility of implementing distributed / parallel processing of Global Rearrangement, the complexity of the task proved to be beyond the scope of this project. For this reason, mpiFITCH does not distribute Global Rearrangement and all characterizations of
mpiFITC performance were made with the Global Rearrangement option turned off. A summary of the default FITC program flow appears in Figure 4.

**Figure 4. The FITC algorithm.** Pseudocode description of FITC (default version that does not execute global rearrangement). A more detailed analysis appears in Addendum 1.

```
Generate initial 3-species tree

While more species remain in distance table

    /* Sequential Addition */
    For each possible position
        ADD next (nth) species to best (n-1) tree
        CALCULATE minimal branch lengths
        IDENTIFY new best n-species tree

    /* Local Rearrangement */
    For each possible position
        SWAP adjacent branches
        CALCULATE minimal branch lengths
        If better tree is IDENTIFIED
            Better tree becomes best tree
            Restart Local Rearrangement

Output final phylogenetic tree
```

**Distributed Design**

**Two Design Options**

Distributed and parallel processing strives for increased efficiency through concurrent, rather than sequential, execution of computational effort. The natural division of FITC tree growth into separable tasks and the iterative nature of each task suggest that both Sequential Addition and Local Rearrangement are candidates for distributed and parallel processing. A straightforward distributed design for Sequential Addition might be to allocate among multiple processors the work of generating and characterizing the \((2n - 5)\) trees during each iteration of tree growth. A simple distribution of the workload might be to assign each processor to examine approximately \((\frac{(2n - 5)}{\text{Cluster Size}})\) topologically distinct trees. (Cluster Size is defined as the number of processors within a cluster that are available for computation; with MPI, all processors can access this value.) Through interprocessor communication, the sum of square values for these trees would be compared, and a new, best n-species tree would be identified. A similar, albeit more complex, design would apply to Local Rearrangement.

With Sequential Addition and Local Rearrangement identified as targets for distributed and parallel processing, many design details remain to be explored. Where will trees be stored? How will dividing the workload be controlled? What information must be transmitted among processors and how frequently must that occur? Two critical parameters related to this last question are data dependency and granularity. Data dependency refers to tasks that must execute in a specific order, as the result of one step cannot accurately be determined without the result of a prior step. If task B is dependent upon task A and both execute in different memory spaces, then the result from task A must be communicated to the processor that is about to execute task B. (At the other extreme, it makes no sense to assign different processors the responsibility of adding different species to the growing tree since it is impossible to execute these tasks in parallel.) Efficiency can be increased by minimizing the distribution of data dependencies. Granularity is a measure of the efficiency of parallel computation and is defined as effort expended on interprocessor communication compared to the amount of work obtained; coarse- and fine-grained parallelism are defined as large and small amounts of computational work, respectively, per unit measure of communication. Minimizing interprocessor communication (by
reducing the size and/or number of transmitted messages) and maximizing the size of concurrent tasks can increase efficiency.

One popular, distributed and parallel design pattern is the master – slave paradigm (Leopold, 2001). In this approach, a master processor makes requests of slave processors, slaves perform assigned tasks and report their results back to the master, and the master makes use the results. A number of phylogenetic programs have been distributed in this manner. Since maximum likelihood methods of tree construction are computationally intense (Hershkovitz and Leipe, 1998), they have often been selected as targets for distributed processing. At least two different master-slave designs have been applied to the distributed / parallel processing of programs based on DNAm (Felsenstein, 1981), a PHYLIP program that iteratively builds trees through Sequential Addition and rearrangement using a maximum likelihood method.

The first of these master-slave designs exchanged large, tree-sized messages among processors (Stewart et al., 2001; Keane et al., 2005); this design was also used by Snell et al. (2000) to build a wrapper program for parallelizing generic phylogenetic software. In this design, the master processor maintained a copy of the growing tree, generated all topologically different n-species trees during Sequential Addition and rearrangement, and transmitted those trees to the slaves. Note that tree transmission requires the exchange of hundreds to thousands of double precision numbers. Upon receiving a tree, a slave calculated branch lengths, returned the analyzed tree to the master, and requested another tree for processing. The master examined all returned trees, identified the best tree, and updated its copy of the growing tree to that best tree (Figure 5A). A second master-slave design pattern used to distribute DNAm (Ceron et al., 1998; Trelles et al., 1998) transmitted small-sized (instructional and small data) messages among processors rather than tree-sized messages. In this approach, master and slaves maintained their own copy of the growing tree. For each task, all processors generated an identical list of subtasks. For example, a list of subtasks for one round of Sequential Addition might include instructions to generate specific trees, or when to calculate branch lengths. The master functioned as subtask dispatcher, assigning items on the list to each slave. Since all processors carried an identical list of subtasks, small messages were sent to slaves as pointer(s) to list. Upon message receipt, a slave consulted and interpreted the list, generated / characterized a specified tree, reported the tree's sum of square value back to the master, and requested additional subtasks. The master compared sum of square values for all the trees that were generated, identified the best n-species tree, requested the slave that generated the best tree to broadcast that information to all the other processors, and all processors updated their copy of the tree to that best n-species tree (Figure 5B).

**Figure 5. Two master-slave design options.** (A) Design pattern utilizing large messages in which trees generated by master are transmitted to slaves. Only master maintains and updates growing tree. (B) Design pattern in which small informational messages are transmitted. Slaves generate and characterize trees, and report Q and position back to master. Once master has identified a very best tree, all processors update their copy of the tree.
Distributed Design of mpiFITCH

When evaluating design options for distributed and parallel processing, the advantages and disadvantages of each must be weighed in light of the particular problem being solved. Despite their design differences, each types of master-slave implementation of DNAml performed well. While considering the applicability of these two designs for the distribution of FITCH, I was concerned that the transmission of tree-size messages would result in poor performance. The reason for this concern related to the difference in computational effort required for branch length determination in likelihood and distance algorithms. Branch length determination for maximum likelihood algorithms (such as DNAml) is computationally intense (a few seconds), making it is well suited for coarse-grained parallelization; any penalty associated with transmitting large messages among processors is dwarfed by the advantage gained by distributing the significant computational effort needed for calculating branch lengths. By contrast, branch length calculations for distance-based algorithms (such as FITCH) are significantly less demanding (fraction of second). If FITCH was designed to transmit large, tree-sized messages while distributing quite modest computational effort, the extent of performance enhancement could be quite low. For this reason, a master-slave design for mpiFITCH was deliberately selected to minimize interprocessor communication and maximize the amount of concurrent, independent computational effort per processor. Minimizing interprocessor communication was accomplished through the use of small-sized messages and by reducing the number of messages (by having slaves assign to themselves their workload). Maximizing the amount of concurrent computational effort was accomplished by having slaves process more than one tree per message. These decisions ultimately required significant alterations to the FITCH code.

If trees are not transmitted, then each processor must maintain a copy of the growing tree. In addition, all processors must update their tree following each Sequential Addition and (perhaps) following Local Rearrangement when a new best or better tree is identified. If trees are not transmitted, then there must be some other way for processors to divide up the workload of generating and characterizing the \((2n - 5)\) and \((2n - 6)\) trees during Sequential Addition and Local Rearrangement, respectively. One way is to divide up the \((2n - 5)\) positions that the next species can be added (or to divide up the \((2n - 6)\) positions that adjacent branches can be swapped). The positions do not have to be assigned to the slaves by a master processor. Instead, each processor can assign to itself a sequential, non-overlapping set of approximately \(( (2n - 5) / \text{Cluster\_Size} \) trees to generate and characterize. How might that work? First, MPI ranks each processor and each processor knows its rank. Second, the positions to add the next species or to swap adjacent branches can be numbered 1, 2, ..., \((2n - 5)\) and each processor knows this order; this count essentially functions as a list of subtasks described by Ceron et al. (1998). For example, if the two processors in a two-processor cluster were to divide up and assign themselves the task of adding the seventh species to the best six-species tree, then based on processor rank and total positions, processor 1 could assign itself to add the seventh species to positions 1 – 5 and Processor 2 could assign itself positions 6 – 9.

For Sequential Addition, this design would be:

1. Based on processor rank, all processors assign to themselves a sequential, non-overlapping set of approximately \(( (2n - 5) / \text{Cluster\_Size} \) topologically different trees to generate / characterize.
2. Each slave adds the next (nth) species to the best (n-1)th tree at each of its assigned \(( (2n - 5) / \text{cluster\_size} \) positions, minimizes branch lengths for each tree, and compares their sum of squares values. At this point, each processor will have identified its own best n-species tree from the subset of \(( (2n-5) / \text{cluster\_size} \) trees that each examined.
3. Slaves inform the master of their best tree. Message #1: \((\text{Cluster\_Size-1})\) messages.
(4) Master compares the sum of square values from each processor’s best n-species tree and identifies a single, ‘best-of-the-best’ n-species tree.

(5) Master informs slave processors the identity (position) of the “best-of-the-best” n-species tree. **Message #2: (Cluster_Size-1) messages.**

(6) All processors update their copy of tree to the best n-species tree.

The two messages must contain the following minimal data:

**Message #1.** To communicate results to the master, Message #1 must specify processor number, the position that the nth species was added to generate that best tree, and the sum of squares value for that tree.

**Message #2.** To inform all processors the identity of the best n-species tree, the master processor must specify the position that the nth species must be added in order to generate that best tree.

**Converting Recursive to Iterative Tree Generation**

Each of the \((2n - 5)\) and \((2n - 6)\) iterations performed during rounds of Sequential Addition and Local Rearrangement, respectively, consists of repeating three subtasks: tree generation, branch length determination, and tree comparison. To distribute these iterations among multiple processors, it was necessary to modify the code responsible for tree generation. Understanding the FITCH tree data structure and the pointer mechanics for species addition and rearrangement required FITCH source code analysis, literature review, and the drawing of (quite literally) hundreds of pictorial representations of trees. This study is summarized Addendum II: Fitch Trees and Their Growth. Note that a FITCH phylogenetic tree is NOT modeled as the classical TREE abstract data type.

**Recursive Tree Generation Is a Poor Model for Balanced Distribution**

The generation of topologically different trees in Sequential Addition and Local Rearrangement is controlled through recursive calls to addtraverse() and/or rearrange(). Each recursive call steps through a linked list that joins the internal vertices of the tree in question, passing pointers to a new tip and new position, or to branches to be swapped. While progression through the linked list is predictable, the order varies with the shape of the tree. *Unfortunately, recursive progression through a linked list is a poor model for equally dividing and distributing a workload among processors.* While linked lists are advantageous for new element insertion and removal, movement to any single (n) element requires \(O(n)\) time. For example, if a linked list of ten positions was divided evenly among five processors \((1,2; 3,4; 5,6; 7,8; 9,10)\), the processor assigned to add the next species at positions 9 and 10 must traverse the list from beginning to end in order to locate the last two positions; the workload among the processors would be unbalanced. Indeed, recursive list traversal further skews the burden. While recursion may simplify the implementation of certain algorithms and enhance their readability, the increased demand it places on the operating system and stack memory often makes it inefficient. In our example, the processor assigned to add the next species at positions 9 and 10 would require more stack space than the processor working on positions 1 and 2. While it might be possible to balance a recursive workload by tuning the division of labor, it is not clear that this would scale well.

**Array-Based Tree Generation Is a Superior Model for Balanced Distribution**

For the reasons described above, equitable load balance is problematic when tasks are arranged as a linked list and the situation is exacerbated by recursive traversal. By contrast, a list of tasks implemented as an array is well suited for balanced division. If the 10 positions that the next species could be added to a tree were in an array rather than a linked list, the time necessary to access any position would be \(O(1)\). Balanced traversal through indexing could easily be adapted...
for distribution among two (or more) processors (for example: for( i = 0; i < 5; i++) and for( i = 5; i < 10; i++) for two processors).

![Diagram of FITCH tree traversal](image)

**Figure 6. Iterative, array-based traversal across a FITCH tree.** (A) Partial representation of a 9-species tree in main memory, immediately prior to the addition of the ninth (J) species. The phylogenetic portion of the tree is represented by the seven sets of triplet nodes (numbered 10 – 16) representing internal vertices. Entry to each set of triplet nodes is through indexed reference to nodep (i.e.: nodep[start] to nodep[end -1]). Traversal down each triplet is by directly stepping through a linked list. (B) Assigning non-overlapping sets of triplet nodes to two processors is accomplished through the assignment of values to variables start and end. (C) The phylogenetic tree represented in (A).

With this in mind, I implemented a predominantly iterative, array-based method of tree generation for Sequential Addition and Local Rearrangement. These new implementations were developed, tested, and verified in a sequential FITCH background run on a single processor PC and were subsequently adapted for use in mpiFITCH. The changes did not require alterations to the FITCH tree data structure. Instead, conversion from recursive to iterative tree generation was by assigning a new functionality to an already existing tree component, the array nodep (Figure 6A). In FITCH, nodep is an array of 'pointers to NODE' that organizes tip vertices and internal vertices prior to their addition to a growing phylogenetic tree. This array normally plays little role in the life of these vertices following their incorporation into the phylogenetic portion of the tree, and plays only a modest accessory role in initiating (but not maintaining) recursive linked list traversal during new tree generation. Converting to a predominantly iterative, array-based traversal was accomplished by using the elements of the nodep array as points of entry to the internal vertices. Traversal across nodep was through standard indexing. Restricting traversal across nodep to the appropriate internal vertices (i.e.: those that were a part of the phylogenetic portion of the tree) required the creation of two new variables, start and end. While access to
each internal vertex was controlled through nodep, the actual interrogation of the three individual
NODEs that composed each internal vertex continued to be performed through direct (not recursive) traversal of a short linked list. Thus, stepwise interrogation of the internal vertices was effected through outer and inner FOR loops controlling access across array nodep (using variables start and end) and the three NODEs of each internal vertex, respectively. Any position resulting in a better tree could be distinctly marked through the counters in the two FOR loops (almost like a 2-D array). This proved a useful means of identifying a NODE that was involved in generating a new best tree, and was used during message exchange and tree update in mpiFITCH. As seen in Figure 6B, balanced division among processors was through the equitable assignment of values to start and end.

Substituting recursive, linked-list traversal of the \((2n - 5)\) positions involved in Sequential Addition with a more iterative, array-based traversal was accomplished by replacing addtraverse() with the new function iterative_addtraverse(). Through code reading combined with pictorial representations, I was able to understand the pointer manipulations responsible for tree generation in Sequential Addition and to convert it from a recursive to an iterative form. Indeed, implementing iterative_addtraverse() required only modest code alterations.

By contrast, implementing array-based Local Rearrangement was extremely challenging. Gaining an understanding of Local Rearrangement through code reading and pictorial representations, an approach that worked so well for Sequential Rearrangement, proved beyond my abilities for Local Rearrangement; the amount of recursion involved in this process was too great to follow. To gain an understanding of the pointer manipulations involved in Local Rearrangement, I next turned to an empirical approach. First, I altered the FITCH code so that it would print out the shapes of all \((2n - 6)\) intermediate trees generated by Local Rearrangement. Second, I analyzed the topologies of these rearranged trees and drew pictures of tree data structures that could possibly represent those trees. Finally, by examining these models, I was able to deduce a set of pointer manipulation rules that accounted for every locally rearranged tree. By combining an implementation of these rules with array-based traversal of nodep, I was able to replace rearrange() with a new function, iterative_rearrange(); the code for iterative_rearrange() was significantly different from rearrange().

The order of tree generation for recursive Sequential Addition and Local Rearrangement differed from the new iterative versions. Order differences should have no effect on Sequential Addition, as all \((2n-5)\) trees are examined. By contrast, order of examination could potentially be important in Local Rearrangement. Recall that if the original (recursive) Local Rearrangement identifies a better tree, then that tree immediately becomes the new best tree, the current round of Local Rearrangement terminates, and a new round of Local Rearrangement begins based upon the new best tree. This introduces two strong data dependencies. Not only does one round of Local Rearrangement significantly influence the results of the next, but also the order that locally rearranged trees are generated and compared seems to be important. The possibility existed that if the order of locally rearranged tree generation and comparison in FITCH and mpiFITCH differed, then final outputs could differ. In an effort to maintain the functionality of Local Rearrangement and attempt to guarantee that FITCH and mpiFITCH produced identical (or at least very similar) output, iterative Local Rearrangement was altered to examine all possible \((2n - 6)\) locally rearranged trees. If one or more better trees were identified, then the better tree with the lowest sum of squares value would be designated the new best tree, and Local Rearrangement would repeat. As seen below, this modification had no effect on the final output tree. (This latter point may not be completely surprising, as an earlier version of FITCH examined all \((2n - 6)\) possible trees (Felsenstein, 1997).)
Effects Of iterative_addtraverse() And iterative_rearrange() In FITCH Background

Performance and output were compared for the original, recursive FITCH (addtraverse() & rearrange()) and for the new iterative FITCH (iterative_addtraverse() & iterative_rearrange()) when executed on a single processor. Iterative tree generation generally examined more trees, reduced runtime by 9%, and had no effect on program output (Table 1).

Table 1
Recursive and Iterative Tree Generation Produce Identical Output on a Single Processor

<table>
<thead>
<tr>
<th>FITCH</th>
<th>Neg</th>
<th>Species</th>
<th>Time</th>
<th>Trees</th>
<th>Q</th>
<th>S.D.</th>
<th>Topology</th>
</tr>
</thead>
<tbody>
<tr>
<td>REC</td>
<td>false</td>
<td>10</td>
<td>&lt; 1</td>
<td>120</td>
<td>0.01907</td>
<td>1.47210</td>
<td>(((F(G((H,E)D))(K(J,B)))C,A)</td>
</tr>
<tr>
<td>ITER</td>
<td>false</td>
<td>10</td>
<td>&lt; 1</td>
<td>118</td>
<td>0.01907</td>
<td>1.47210</td>
<td>(((B,J)K)((G(D(E,H)))F))C,A</td>
</tr>
<tr>
<td>REC</td>
<td>false</td>
<td>50</td>
<td>90</td>
<td>8760</td>
<td>1.55906</td>
<td>2.52363</td>
<td>N.S.</td>
</tr>
<tr>
<td>ITER</td>
<td>false</td>
<td>50</td>
<td>82</td>
<td>8902</td>
<td>1.55906</td>
<td>2.52363</td>
<td>N.S.</td>
</tr>
<tr>
<td>REC</td>
<td>false</td>
<td>75</td>
<td>550</td>
<td>20909</td>
<td>3.84716</td>
<td>2.63331</td>
<td>N.S.</td>
</tr>
<tr>
<td>ITER</td>
<td>false</td>
<td>75</td>
<td>501</td>
<td>22157</td>
<td>3.84716</td>
<td>2.63331</td>
<td>N.S.</td>
</tr>
</tbody>
</table>

REC, addtraverse() & rearrange() [the original version of FITCH]
ITER, iterative_addtraverse() & iterative_rearrange()
Neg, negative branch lengths permitted
Species, total number of species in final tree
Time, seconds
Q, sum of squares
S.D., average percent standard deviation
Topology, tree represented in Newick format. Note that the while the two Newick strings for the 10-species trees are written slightly differently, they represent identical trees.
N.S., not shown

Distributed Implementation

The mpiFITCH functions involved in Sequential Addition and Local Rearrangement appear below. Function mpi_add_next_species_to_tree() controls the workflow of distributed Sequential Addition through calls to four additional functions; mpi_non_rec_rearrange() controls the workload of distributed Local Rearrangement.

Sequential Addition
mpi_add_next_species_to_tree()
  mpi_divide_the_workload()
  mpi_slaves_add_next_species_to_their_restricted_part_of_tree()
  myOp_where_best_to_add_next_species()
  mpi_all_processors_update_by_adding_next_species_to_the_very_best_position()

Local Rearrangement
mpi_non_rec_rearrange()
  mpi_divide_the_workload()
  mpi_slaves_non_rec_rearrange_on_restricted_part_of_tree()
  mpi_rearrange_the_nodes()
  mpi_insert_for_non_recursive_rearrangement()
  myOp_where_best_to_non_rec_rearrange()
  mpi_all_processors_update_by_non_rec_rearrangement()

Dividing Up the Workload: Self-Assignment Based On Processor Rank

Dividing up the workload of tree generation was accomplished by each processor assigning to itself non-overlapping values to variables start and end (Figure 6A,B) based on cluster_size,
cluster_rank, and the number of nodep elements to be traversed (nodes_to_traverse). The algorithm, in function mpi_divide_the_workload(), does not guarantee exact load balance; however it averages out well and handles cases where the workload divides equally or unequally among processors. The assigning code is presented below and load balancing is presented for three conditions (Table 2).

```c
int step = nodes_to_traverse / cluster_size;
int remainder = nodes_to_traverse % cluster_size;

if( cluster_rank < remainder ){
    *start = ( cluster_rank * step ) + cluster_rank + numsp;
    *end = *start + step + 1;
}
else {
    *start = ( cluster_rank * step ) + remainder + numsp;
    *end = *start + step;
}
```

Table 2
Load Balancing Among Processors for a 50 Species Tree

<table>
<thead>
<tr>
<th>Nodes To Traverse</th>
<th>Cluster Size</th>
<th>step</th>
<th>remainder</th>
<th>Rank 0</th>
<th>Rank 1</th>
<th>Rank 2</th>
<th>Rank 3</th>
<th>Rank 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>0</td>
<td>4</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>54</td>
</tr>
<tr>
<td>17</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>50</td>
<td>54</td>
<td>58</td>
<td>61</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>54</td>
<td>58</td>
<td>61</td>
<td>64</td>
<td>67</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>50</td>
<td>54</td>
<td>58</td>
<td>62</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>54</td>
<td>58</td>
<td>62</td>
<td>66</td>
<td>70</td>
</tr>
</tbody>
</table>

Rank, processor number
numsp, total number of species in tree (here, 50)

Iterative Tree Generation
With the validity of iterative, array based tree generation confirmed in a FITCH background (Table 1), it was relatively straightforward to incorporate these functions into mpiFITCH to support distributed and parallel tree generation and branch length determination. For this, iterative_addtraverse() was reworked into function mpi_slaves_add_next_species_to_their_restricted_part_of_tree() and iterative_rearrange() was reworked as function mpi_slaves_non_rec_rearrange_on_restricted_part_of_tree(). Each processor generates its assigned subset of new trees, determines optimal branch lengths for that subset by minimizing sum of square values, and identifies a new best or (perhaps) better tree from that subset.

Message Transmission Using MPI
At this point, for Sequential Addition every processor assigned a portion of nodep will have identified a new best n-species tree; for Local Rearrangement processors may or may not have identified a better tree. To identify a single best-of-the-best tree, all of the best trees identified by the slave processors must be transmitted to the master for comparison. Interprocessor communication is accomplished through MPI library functions.
Point-To-Point Message Passing: MPI_Send() and MPI_Receive(). The simplest MPI method of message passing is point-to-point transmission using paired MPI_Send() and MPI_Receive() function calls. In the code below, the master receives a message from each slave one at a time. For each received message, the sum of square value is compared to that of the previous best tree. If a better tree is identified, its identifying characteristics (position along nodep) is saved in very_best_data.

```c
if(cluster_size > 1) //if more than one processor in cluster
{
    if( cluster_rank > 0 ) //if a slave processor, then send
        MPI_Send(best_data, 5, MPI_DOUBLE, 0, 12,
                 MPI_COMM_WORLD);
    else if( cluster_rank == 0 ) //master processor
    {
        //Master receives from slaves, one at a time.
        for(k = 1; k < cluster_size; k++)
        {
            MPI_Recv(reply_from_slave, 5, MPI_DOUBLE,
                      MPI_ANY_SOURCE, 12, MPI_COMM_WORLD, &status);
            //Compare sum of square value
            //Save slave tree that is better
            if(reply_from_slave[0] > very_best_data[0])
            {
                very_best_data[0] = reply_from_slave[0];
                very_best_data[1] = reply_from_slave[1];
                very_best_data[2] = reply_from_slave[2];
                very_best_data[3] = reply_from_slave[3];
            }
        }
    }
}
```

Once the master has examined the messages from all slaves and identified a new best tree, it informs all processors of the unique position to add or swap in order to generate that tree (thus enabling each processor to update their own tree). This information is transmitted to the slaves through point-to-point MPI_Send() and MPI_Receive() calls.

```c
if( cluster_rank == 0 )
    for( k = 1; k < cluster_size; k++ )
        MPI_Send( very_best_data, 5, MPI_DOUBLE, k, 13,
                  MPI_COMM_WORLD);
else if( cluster_rank > 0 ) //Slaves receive very best data
    MPI_Recv( very_best_data, 5, MPI_DOUBLE,
             MPI_ANY_SOURCE, 13, MPI_COMM_WORLD, &status );
```

Collective Communication: MPI_Allreduce(). In the above code, multiple MPI_Send() / MPI_Receive() messages were coded explicitly by the programmer within a FOR loop and the bulk of the work was restricted to the master. This approach is associated with a certain amount of inefficiency and is error prone. For example, while the master processor is receiving a message from slave processor 3, or while the master processor compares sum of squares values, all other processors are idle. In both cases, some processors are waiting rather than computing!

MPI collective communication routines encapsulate one-to-all, all-to-one, and all-to-all interprocessor communication into single function calls. This relieves the programmer of
explicitly coding multiple point-to-point calls and reduces error (Pacheco, 1997). Certain, but not all, implementations of MPI collective communication routines can be tuned during MPI library development and at runtime to take advantage of specific system hardware architecture (network topology, switches) to support concurrent interprocessor communication (Faraj and Yuan, 2005; Gorlatch, 2004). If these conditions are met, (and they generally are not), transmission time could be reduced from \((\text{Cluster Size} - 1)\) to a minimum of \(\lceil \log_2 (\text{Cluster Size}) \rceil\). Besides easing message passing, collective communication also supports built-in and user-defined computation. \(\text{MPI\_Allreduce()}\) supports all-to-all communication. With the help of a user-defined function to perform sum of square comparison, this single line of code

\[
\text{MPI\_Allreduce}(\text{best\_data, very\_best\_data, 5, MPI\_DOUBLE, *myOp, MPI\_COMM\_WORLD});
\]

(with the assistance of a user-defined helper function pointed to by myOp) can replace all the code presented in the previous section ("Point-To-Point Message Passing: MPI\_Send() and MPI\_Receive()"). Indeed, the ability of \(\text{MPI\_Allreduce()}\) to transmit and to compute eliminates the need by mpiFITCH for a designated master processor; all processors can function as slaves.

**Update**

Once processors are informed of the best position to add the next species or to swap adjacent branches, they use that information to immediately update their own copy of the tree. Processors move pointers directly to the correct position and use functions similar to those used for tree generation and branch length determination to update to the correct tree. At this point, all copies of best tree are identical.

**Performance**

Runtime performance of mpiFITCH was characterized on two Beowulf clusters. Each cluster was constructed and is maintained by R. Zaritski (Montclair State University). Galaxy (GAL) is a 20-processor, Pentium-based (1 GHz) cluster (slave RAM 256 MB, 133 MHz) running Scyld Beowulf Cluster Operating System (Linux-based) and connected by a Switched Gigabit Ethernet over Cat 5e/6 copper cables with a 12-Port Gigabit SuperStack-3 4900 switch (http://roman.montclair.edu/Research/Parallel/Galaxy). Parallel Monster (PM) is a 64-processor Opteron-based (1.4 GHz) cluster (slave RAM 2 GB, 333 MHz) running Red Hat Linux and connected by a Switched Gigabit Ethernet over Cat 5e/6 copper cables with a 36-Port HP 4108GL switch (Zaritski and Pal, 2006; http://roman.montclair.edu/Research/Parallel/PM).

Timed runs of the original sequential FITCH on GAL and PM were approximately 5.4 hr and 2.8 hr, respectively, for 200-species trees, making PM approximately two times faster than GAL. The data set used for testing mpiFITCH was derived from genotype analysis of over 250 North and South American strains of maize, each characterized for microsatellite repeats at 99 loci (Matsuoka et al., 2002). I developed a small C++ program (msr_distance.cpp) that converted the raw genotype data (generously provided by C. Du, Montclair State University) to pairwise distances arranged as a distance table using Ps (proportion of alleles shared averaged of loci) as a measure of similarity and D (distance) as a measure of divergence (Bowcock et al., 1994); \(Ps = \frac{\text{number of shared alleles summed over loci}}{2 * \text{number of loci compared}}\) and \(D = 1 - Ps\).

Performance was characterized by run time, speed up, efficiency, and characteristics of the final tree. All tests were performed with the Global Rearrangement option turned off. Speedup(P) for P processors is defined as \(\frac{T_1}{T(P)}\) where \(T_1\) is the run time of a sequential version of the program (using a single processor) and \(T(P)\) is the run time using P processors; efficiency(P) for P processors is \(\frac{\text{speedup(P)}}{P}\) (Leopold, 2001). Function main() was modified to time program runs and to write runtimes, Q, average percent standard deviation and number of trees examined.
into a file named statsfile. The distributed program mpiFITCH (which was implemented using iterative versions of both Sequential Addition and Local Rearrangement, and MPI_Allreduce() for message passing) exhibited good performance. On both clusters, execution on multiple processors had no effect on the sum of square and average percent standard deviation for the final output tree (Table 3). In addition, cursory inspection (spot check) of the output trees themselves revealed them to be the same (data not shown). On the PM cluster, mpiFITCH showed a ~24 – ~28 fold speed up (Figure 7A,B) using the maximum number of processors (64), with an efficiency of ~ 40% (Figure 7C). Importantly, at no point did the rate of speed up level off or fall. Speed up on the GAL cluster was slightly less efficient than on PM (Figure 7D).

![Graph A](image)

![Graph B](image)

![Graph C](image)

![Graph D](image)

**Figure 7.** mpiFITCH performance on clusters. (A) Representative run time, (B) speed up, and (C) efficiency of mpiFITCH on PM cluster using up to 64 processors. (D) Representative comparison of mpiFITCH speed up on PM and GAL clusters (225 species tree).

**Table 3**

<table>
<thead>
<tr>
<th>Cluster</th>
<th>PM</th>
<th>Processors</th>
<th>Neg</th>
<th>Species</th>
<th>Time</th>
<th>Trees</th>
<th>Q</th>
<th>S.D.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FITCH</td>
<td></td>
<td>1</td>
<td>true</td>
<td>50</td>
<td>24</td>
<td>7876</td>
<td>1.49011</td>
<td>2.46719</td>
</tr>
<tr>
<td>mpiFITCH</td>
<td></td>
<td>1</td>
<td>true</td>
<td>50</td>
<td>24</td>
<td>6468</td>
<td>1.49011</td>
<td>2.46719</td>
</tr>
<tr>
<td>mpiFITCH</td>
<td></td>
<td>4</td>
<td>true</td>
<td>50</td>
<td>12</td>
<td>6468</td>
<td>1.49011</td>
<td>2.46719</td>
</tr>
<tr>
<td>mpiFITCH</td>
<td></td>
<td>1</td>
<td>true</td>
<td>50</td>
<td>56</td>
<td>6468</td>
<td>1.49011</td>
<td>2.46719</td>
</tr>
<tr>
<td>mpiFITCH</td>
<td></td>
<td>4</td>
<td>true</td>
<td>50</td>
<td>20</td>
<td>6468</td>
<td>1.49011</td>
<td>2.46719</td>
</tr>
</tbody>
</table>

Abbreviations as in Table 1.
PM, Parallel Monster; GAL, Galaxy Cluster
Architectures that support concurrent message passing can be engaged through the use of MPI collective communication routines and this may reduce transmission time. By comparing versions of mpiFITCH that communicate through MPI_Send()/MPI_Receive() and MPI_Allreduce() it was possible to detect any performance advantage. MPI_Allreduce() with multiple processors failed to enhance performance on both PM (Figure 8A) and GAL (data not shown) clusters. This suggests that either the hardware elements of these clusters, and/or the MPI libraries used, were not able to support concurrent message passing. The former rationale is supported by the relatively inexpensive Ethernet switches used on both the GAL and PM clusters.

Figure 8. Performance of mpiFITCH variants. (A) Comparison of point-to-point and collective communication within two versions of mpiFITCH revealed no advantage of MPI_Allreduce on the PM cluster (200 species tree). (B) A version of mpiFITCH (rec_mpiFITCH) that distributed Local Rearrangement by retaining recursive linked list traversal exhibited poor performance on PM and GAL clusters (200 species tree). Representative results

Another distributed implementation of FITCH, rec_mpiFITCH, attempted to distribute Local Rearrangement while retaining the identical Local Rearrangement algorithm used by sequential FITCH. By this I mean that positions to swap adjacent branches were identified by recursive traversal through a linked list. When tested, this implementation (distributed, recursive Local Rearrangement combined with distributed, iterative Sequential Addition) exhibited poor performance. Maximum speed up reached ~2.5 fold and ~ 4.5 fold with an intermediate number of processors on GAL and PM, respectively, and decreased as more processors were added (Figure 8B). This demonstrates the adverse effect brought on by linked list traversal with recursion.

Lastly, I compared performance among distributed implementations of a number of different phylogenetic programs. The value of such a comparison is tenuous, particularly when different tree-building methods are being evaluated. Nonetheless, comparison of maximum speed up for mpiFITCH and a number of other distributed implementations for distance, parsimony, and likelihood software that were mentioned in this text suggests that mpiFITCH exhibits relatively good performance (Table 4).
<table>
<thead>
<tr>
<th>Method</th>
<th>Processors</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpiFITCH Distance</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>mpiFITCH Distance</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>mpiFITCH Distance</td>
<td>32</td>
<td>17</td>
</tr>
<tr>
<td>mpiFITCH Distance</td>
<td>64</td>
<td>27</td>
</tr>
<tr>
<td>Snell et al. (2000) PR</td>
<td>32</td>
<td>9</td>
</tr>
<tr>
<td>Du and Lin (2005) MP</td>
<td>10</td>
<td>4 – 4.5</td>
</tr>
<tr>
<td>Keane et al. (2005) ML</td>
<td>60</td>
<td>25</td>
</tr>
<tr>
<td>Ceron et al. (1998) ML</td>
<td>64</td>
<td>36</td>
</tr>
<tr>
<td>Stewart et al. (2001) ML</td>
<td>64</td>
<td>50</td>
</tr>
</tbody>
</table>

ML, maximum likelihood; MP, maximum parsimony; PR, parallel ratchet; mpiFITCH, on PM cluster (250 species tree)

**Conclusions**

FITC H is a distance-based, evolutionary tree-building program designed for sequential execution on a single processor. This study designed and implemented a modified version of FITC H, mpiFITC H, that incorporates MPI message passing to facilitate distributed and parallel execution. Thus far, mpiFITC H distributes two of the three major computational components of FITC H, Sequential Addition and Local Rearrangement. Performance of mpiFITC H on a 64-processor cluster revealed maximum speed up and efficiency of ~28 fold and ~40%, respectively. To support this level of performance, it was necessary to replace the recursive nature of Sequential Addition and Local Rearrangement with iterative versions that were more conducive to efficient distribution. While MPI collective communication functions offered no advantage over point-to-point communication on the two Beowulf clusters tested, additional speed up may be possible with clusters supporting concurrent message passing. Future studies will focus on implementing a distributed version of the third major component of this program, Global Rearrangement.
References


Addendum I: FITCH Program Flow

BEGIN
Read in distance matrix from data file
Generate initial 3-species tree and designate as best tree

While there are species remaining to be added to the tree
Loop

/* Sequential Addition */
For each of \((2n - 5)\) positions that the next \((n)\)th species can be added to the best \((n - 1)\) - species tree
Loop
   Add next species at that position to generate new \(n\)-species tree
   Minimize branch lengths for the new tree
   If this is the first \(n\)-species tree examined
      Designate new tree as best \(n\)-species tree
   Else if sum of squares for new tree is less than that of best \(n\)-species tree
      Designate new tree as best \(n\)-species tree
End For Loop

/* Local Rearrangement */
For each of \((2n - 6)\) Local Rearrangements that can be made to the best \(n\)-species tree
Loop
   Swap adjacent branches to generate a new \(n\)-specie tree
   Minimize branch lengths for this new tree
   If sum of squares for new tree is less than that of best \(n\)-species tree
      Designate new tree as best \(n\)-species tree
      Initiate new round of Local Rearrangement
End For Loop

/* Global Rearrangement */
If last species has been added to tree and global option was selected
   For each possible Global Rearrangement that can be made to best \(n\)-species tree
      Loop
         Remove and reinsert subtree to generate new \(n\)-specie tree
         Minimize branch lengths for this new tree
         If sum of squares for new tree is less than that of best \(n\)-species tree
            Designate new tree as best \(n\)-species tree
            Initiate new round of Global Rearrangement
      End For Loop
End While Loop

Output phylogenetic tree

END
Starting with a unique 3-species tree, FITCH generates trees by iterative tree growth. Each iteration of tree growth (defined as passage through the WHILE loop in the above pseudocode) is divided into two major tasks, Sequential Addition and Local Rearrangement. The best tree identified by an iteration serves as starting point for the next iteration. Following the addition of the last species, an optional third major task, Global Rearrangement, may be performed. For increased clarity, Local and Global Rearrangement are explicitly separated in the above pseudocode; a more realistic rendition would reveal that Global Rearrangement executes in intimate association with the final round of Local Rearrangement.

Addendum II: FITCH Trees and Their Growth

Abstract Tree
A tree is mathematically defined as a connected, acyclic, undirected graph. For our purposes, a phylogenetic tree can be thought of as being composed of internal and tip (leaf) vertices connected by branches. All vertices can be thought of as representing a taxon (a taxonomic unit), the most common being a species (a population of interbreeding individuals). Tip vertices are incident on one branch and generally represent living species; all other tree positions represent a time in the past. In most cases, speciation (the generation of new species) is thought to occur through one species giving rise to two. Because of this, most phylogenetic trees are bifurcating. Internal vertices represent a past, most recent common ancestor; internal vertices are incident on three branches. Divergence from an internal vertex represents the accumulation of sufficient genetic differences to preclude interbreeding and to warrant classification as two separate species. Phylogenetic trees are commonly thought of as being rooted and exhibiting directionality. Unrooted phylogenetic trees are often easier to construct and can be converted to a rooted tree through the identification of an outgroup (a distant species). The final output from FITCH is an unrooted, bifurcating tree.

A computer representation of a tree often uses a structure NODE to represent a vertex. At least one member of NODE is of type ‘pointer to NODE.’ When the value of a node’s ‘pointer to NODE’ is NULL, the node may be unattached to a tree or it may be a tip. When the value of a first node’s ‘pointer to NODE’ is the address of a second node, the address functions as a branch connecting the two nodes.

Data Structures of a FITCH Tree
A FITCH tree is significantly more complex than the familiar tree data structure used by computer scientists and programmers. Rather, a FITCH is composed of three distinct, yet associated, data structures: one TREE structure, one array of ‘pointer to NODE,’ and at least six (and usually many more) NODE structures.

The first component of a FITCH tree, the TREE structure, is defined as:

```c
struct TREE
{
    struct NODE ** nodep;
    double likelihood;
    struct NODE * start;
};
```

TREE member nodep associates a TREE structure with an array of ‘pointer to NODE’ by pointing to the first element of that array; likelihood reflects a measure of how well a specific
tree (topology and branch lengths) models a set of experimental data and is related to the tree’s sum of square value; start is a pointer to the first species added to the tree.

The second component of a FITCH tree is an array of ‘pointer to NODE.’ While nodep links TREE to this array, each element in the array (nodep[0], nodep[2], etc.) points to a NODE.

The third component of a FITCH tree, NODE, is defined as:

```c
struct NODE
{
    struct NODE *next, *back;
    boolean tip, iter;
    short number;
    char name[10];
    double *d, *w;
    double v, dist;
    short xcoord, ycoord, ymin, ymax;
};
```

The two types of verticies in a FITCH tree are represented by different arrangements of NODEs. A tip (leaf) vertex that symbolizes a living species is represented by a single NODE. By contrast, an internal vertex in a FITCH tree is represented by a circular arrangement of three connected NODEs. This unexpected layer of complexity actually simplifies species addition and branch swapping during tree growth and rearrangement, respectively.

Memory Allocation: A Branchless and Leafless FITCH Tree
Consider a tree of \( z = 5 \) species. When complete, the final unrooted tree will consist of five tip verticies representing the five living species, three internal verticies, and seven branches (two of which are internal). Upon initialization, the data structures allocated in memory for the branchless and leafless tree include one TREE structure, one array of ‘pointer to NODE’ containing \((2z - 2) = (2 * 5 - 2) = 8\) elements, and \((4z - 6) = 14\) NODEs (Figure 9). The first five NODEs (pointed to by the first five elements of nodep) represent the five species in the distance matrix; at this time, member values for the unconnected species NODEs are initialized to tip = true; next = NULL; back = NULL; number = 1,2,...,z (first species, second, etc); nayme stores the species name (here A – E); d and w point to arrays of \((2z - 2)\) doubles. The remaining three elements of the nodep (nodep[5] through nodep[7]) each point to one NODE of a circular triplet of NODEs. Each (vertical) triplet set represents one internal vertex. At this time, member values for these NODEs are tip = false; next (a ‘pointer to NODE’) maintains the circular nature of the triplet set of NODEs; number = \((z + 1),...,2z - 2\); d and w point to arrays of \(2z-2\) doubles. At this point, the TREE is composed of a region of unconnected species NODES, and a region of unconnected internal verticies.

The Initial, Three-Species Phylogenetic Tree
To generate the initial three-species phylogenetic tree, the first three species (NODEs 1, 2, and 3) become associated through branches with the first internal vertex (the circular set of triplet NODEs numbered 6). Branches between tips and the single internal vertex are generated by assigning values (addresses) to NODE member back (FIGURE 10A). For any three-species unrooted tree, a single topology is possible and its branch lengths can be solved exactly. At this point, the TREE consists of three regions, the phylogenetic tree (three species connected to an internal vertex, two unlinked species (D and E), and two unlinked internal verticies (7 and 8).
Figure 9. **Initialized tree in memory.** Diagrammatic representation of a 5-species FITCH tree in main memory prior to building any phylogenetic tree. Species (leaf) vertices are composed of single NODEs named A – E and numbered 1 – 5. The three internal vertices that will be a part of the final tree are represented by three vertical sets of three circularly connected NODEs (numbered 6 – 8). One tip vertex and one internal vertex are indicated.

Figure 10. **The three-species FITCH tree.** (A) Main memory representation of the first phylogenetic tree to be generated, linking leaf vertices A, B, and C with a single internal vertex (the set of three interconnected NODEs numbered 6). (B) Simplified (upper) and stick-figure (lower) representations of the same three-species phylogenetic tree depicted in (A).
Simplified Representation of a FITCH Tree

FITCH trees are difficult to draw. To assist myself in depicting larger trees, I developed a simple representation of a FITCH tree. This proved to be a tremendous asset as I attempted to understand the process by which species are added to, and rearranged on, these trees. A simplified version of the three-species phylogenetic tree in Figure 10A is presented in Figure 10B, along with the stick-figure tree that it represents. Complexity was reduced by eliminating all data structures except for NODEs. A vertical, circular, triplet set of NODEs in Figure 10B represents a circular triplet set of NODEs in Figure 10A (one internal vertex); tip verticies are shown as species names and are connected to an internal vertex through back pointers (black lines).

FITCH Tree Growth by Sequential Addition

Tree growth by the addition of one species requires that a new tip, a new internal vertex, and a new branch be added to the tree. The first step in adding the next (nth) species to a (n-1) tree is to connect the next species to the next internal vertex. Continuing our example, adding the fourth species (species D, NODE 4) to the three-species tree is begun by associating the species D NODE with the next unused triplet set (NODEs 7), creating a new tip (Fig11A). The fourth species can be added to a 3-species tree at (2n - 5) = 2 * 4 - 5 = 3 different positions. To represent the 4-species tree in which species A and D are adjacent, species A (NODE 1) is unlinked from its internal vertex (triplet set NODE 6) and immediately relinked to either one of the two free NODEs of the newest internal vertex (triplet set NODE 7) (Figure 11B). Finally, the remaining two free NODEs are connected through their ‘back’ pointers (Figures 11C,D). A summary of the arrangements of the other two topologically distinct, 4-species trees appears in Figure 12. To identify which of these three trees best fit the experimental data, FITCH would minimize branch lengths for all three trees and compare their sum of squares values. The best 4-species tree identified by sequential addition would then be subjected to local rearrangement.

Adjacent Branch Swapping by Local Rearrangement

One possible five-species tree appears in Figure 13A. Local rearrangement of this tree can generate (2n – 6) = 4 distinct topologies, two of which appear in Figures 13B and 13 C. Note that in these two cases, branch B was moved so as to be adjacent to species C (Figure 13B) and...
species E. The other two rearranged trees would have moved branch B adjacent to species A and D.

Figure 13. Adjacent branch swapping by local rearrangement. Adjacent branch swapping of the 5-species tree in (A) results in four new trees, two of which are shown in (B) and (C).