ExactMP: An Efficient Parallel Exact Solver for Phylogenetic Tree Reconstruction Using Maximum Parsimony

David A. Bader [*] Vaddadi P. Chandu Mi Yan
College of Computing Linux Networks Inc.
Georgia Institute of Technology

February 26, 2006

Abstract

Constructing phylogenetic trees in the study of the evolutionary history of a group organisms is an extremely challenging problem in computational biology. The problem becomes intractable with growing number of organisms. In this paper, we design and implement an efficient parallel solver (ExactMP) using a parsimony based approach for solving this problem. We create a testbed consisting of eighteen datasets of varying size (up to 27 taxa) and difficulty level (easy to hard), containing real (Eukaryotes, Metazoan, and rbcL) and randomly-generated synthetic genome sequences. We demonstrate our ExactMP Solver against this testbed and achieve a parallel speedup of up to 7.26 with 8 processors using an 8-way symmetric multiprocessor. The main contributions of this work are: (1) an efficient parallel solver ExactMP for the problem of phylogenetic tree reconstruction using maximum parsimony, (2) a new upper bounding methodology for this problem using heuristic and randomization techniques, and (3) a highly optimized branch and bound algorithm for this problem.

1 Introduction

Knowledge of accurate evolutionary information among organisms is essential for classification, taxonomy, and molecular epidemiological study of viruses. Evolutionary history is

[*]This work was supported in part by NSF Grants CAREER CCF-0611589, ACI-00-93039, NSF DBI-0420513, ITR ACI-00-81404, DEB-99-10123, ITR EIA-01-21377, Biocomplexity DEB-01-20709, and ITR EF/BIO 03-31654; and DARPA Contract NBCH30390004.

represented by a labeled acyclic graph known as a *phylogenetic tree* whose leaf nodes represent the known organisms and internal nodes represent hypothetical ancestral organisms. Construction of such a tree requires a dataset containing a group of organisms (also known as taxa), associated states representing the characteristic features of each organism in the group (also known as characters or sites), and an optimality criterion to establish the relationship among those organisms.

Methods to construct a phylogenetic tree can be classified as heuristic (an approximate solution) [28, 30, 11, 20, 29, 23], and exact (where all possible tree topologies are evaluated according to some optimality criterion) methods. For both methods, Maximum Parsimony (MP) and Maximum Likelihood (ML) are the two most popularly known optimality criteria. For a given dataset, ML criteria assumes a stochastic model and selects a phylogenetic tree that gives rise to the data with highest probability. MP criteria, on the other hand, selects the phylogenetic tree with fewest number of evolutionary changes. In this paper, we will focus on MP criteria.

1.1 Maximum Parsimony Criterion

In the Maximum parsimony (MP) criterion, an accurate phylogenetic tree is selected in three steps: (a) all possible tree topologies are enumerated, (b) each tree topology is assigned a score based on some parsimony strategy, and (c) from a set of all such scored trees, the tree with lowest score is selected as the *optimal* solution.

Many parsimony strategies, such as Fitch [8], Farris [7], Swofford et al. [31], are proposed in the literature for use with the MP method. In this paper, for simplicity we focus on Fitch's strategy. However the techniques we use are applicable to all of these strategies.

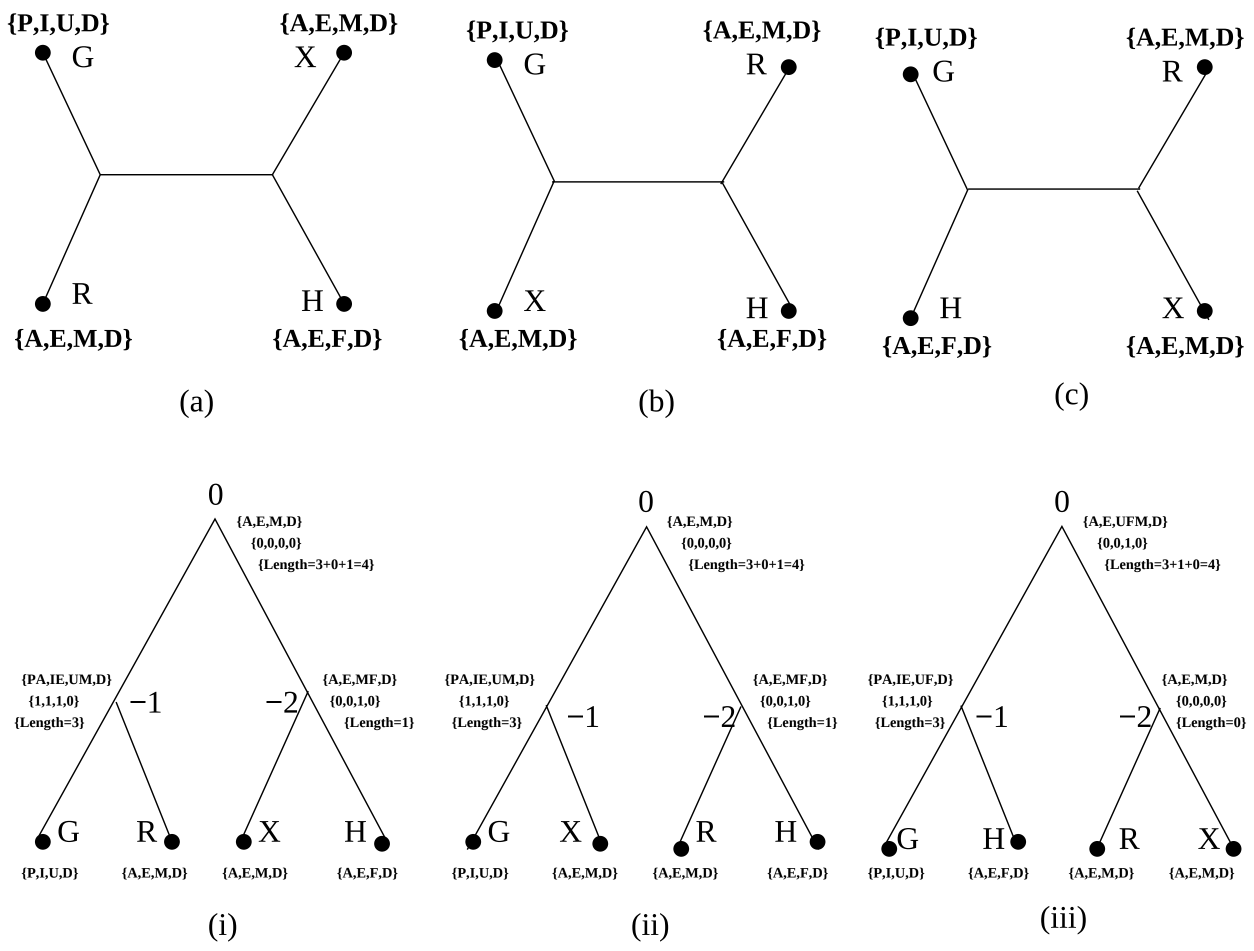

Figure 1: MP analysis on dataset shown in Table 1.

Fitch's strategy is a two-pass method for scoring a particular topology of a phylogenetic tree and is based on the assumption that each character (characteristic feature) evolves independently. It takes as an input, a phylogenetic tree topology with m organisms and a set of all possible characters. If a state set S of an organism be defined as the set containing all its characteristic features, then Fitch's first pass can be described as follows: starting from the leaf nodes and working its way up to an arbitrary root node in a given tree topology, (a) assign a state set to each internal node in the tree, and (b) simultaneously compute the score of the tree. Fitch's second pass optimizes the state sets computed in part (a). The algorithm can be stated as follows:

1. To each leaf node, assign a state set containing all the characters of the corresponding organism and set the *score* of the tree to 0.

2. For each internal node S_{parent} and its children S_{left_child} and S_{right_child}, at each site

   a. $S_{parent} = S_{left\_child} \cap S_{right\_child}$.

   b. If $S_{parent} = 0$, then

      $S_{parent} = S_{left\_child} \cup S_{right\_child}$ and

      increment the *score of the tree* by 1.

For each union operation the score of the phylogenetic tree increments by one; thereby making it less parsimonious. Once the *state_sets* of all internal nodes are computed, if an optimal state assignment is desired, a second pass may be applied as described in [8]. In phylogenetic tree reconstruction, we are primarily interested in the tree score and not in an optimal state assignment of the internal nodes. Hence, we safely ignore running the second pass on each candidate topology and apply it only once for the final optimal tree. Fig. 1.1 illustrates the MP analysis on a dataset containing four suborders of beetles [1] shown in Table 1. The three possible tree topologies are shown in (a), (b), and (c). (i), (ii), and (iii) show the Fitch's score of (a), (b), and (c), respectively. Node 0 represents the arbitrary root node. Nodes G, R, X, and H represent Polyphaga, Archostemata, Myxophaga, and Adephaga respectively. Nodes with negative numbers are the inferred ancestors. Alphabetic characters represent the states in that site and numeral values represent the score at the site. In this case, the tree score for all topologies is 4. Hence, this dataset has 3 optimal solutions, each of score 4.

	Cervical sclerites	**Propleuron**	**Hind Coxae**	**First Visible Abdominal Sternum**
Polyphaga (G)	Present (P)	Internal (I)	Usually movable (U)	Undivided (D)
Archostemata (R)	Absent (A)	External (E)	Movable (M)	Undivided (D)
Myxophaga (X)	Absent (A)	External (E)	Movable (M)	Undivided (D)
Adephaga (H)	Absent (A)	External (E)	Fused (F)	Undivided (D)

Table 1: A dataset containing 4 suborders of beetle, each with four characteristic features. Rows represent the organisms and columns represent their features.

2 Exact Methods

Methods to generate exact phylogenetic trees consist of Exhaustive and Branch & Bound approaches. For $D_{n,m}$ (n taxa and m characters or sites, which we refer to as $D_{n,m}$), the exhaustive method searches for a tree with lowest score from all $\Pi_{i=n}^{3}(2i-5)$ trees. Since $\Pi_{i=n}^{3}(2i-5)$ grows rapidly (doubly-factorial) with n, exhaustive search is intractable for even datasets of modest size. For problems of this nature, a generalized approach is to use Branch & Bound (B&B) [12]. B&B search is a general technique for solving combinatorial optimization problems of high complexity where exhaustive search is intractable. A general B&B algorithm can be viewed as an enumeration method for an optimization problem p : $Z(p) = min_{x \epsilon X}\ F(x)$ where X represents the domain of the problem p, and x is a solution. x is feasible iff $x \in X$, $F(x)$ is the cost of the solution, also called the objective cost. The underlying idea of the B&B algorithm is successive decomposition of the original problem into smaller disjoint subproblems until one or all optimal solutions are found.

A B&B search tree can be modeled as a rooted tree T with a cost function over its leaves. The goal is to find the leaves with minimum cost in T. Nodes of T represent subproblems and edges from nodes to their successors represent the decomposition of a problem into subproblems. Input to the algorithm is the root of tree T. B&B search involves four basic operations: Subproblem Selection, Tree Traversal, Subproblem Generation, and Termination, each of which is governed by the following four rules: Selection rule, Branching rule, Elimination rule, and Termination rule, respectively. Branching rule divides a feasible solution set X into X_1, X_2, $\ldots$, X_n, where $x \in_{i=1}^{n} X_i$ and $X_i \cap X_j = \emptyset$ for all $i \neq j$. Let p_i denote the optimization subproblem corresponding to X_i, and $Z(p_i)$ be its optimal value, then, $Z(p) = \min_{1 \leq i \leq n} Z(p_i)$. Selection Rule chooses the most promising subproblem for further branching, and Elimination Rule recognizes and eliminates subproblems that cannot yield an optimal solution. Termination Rule determines whether a complete solution is optimal or not. A subproblem Q can be eliminated if (a) Q has been solved or (b) there exists another subproblem R which dominates Q, i.e., $Z(R) \leq Z(Q)$. For any B&B algorithm, branching and termination rules are problem specific, while selection and elimination rules rely on search strategies and data structures.

2.1 Branch & Bound Approach for MP

This section presents our B&B approach to solve the problem of phylogenetic tree reconstruction and describes our novel techniques to improve its performance. First we introduce the terminology that we will be using in rest of the paper. For a dataset $D_{n,m}$, (a) any subproblem T_k^q is a *partial tree* with k taxa at q^{th} level in the B&B search tree, for $3 \leq k < n$, and $1 \leq q \leq (n-2)$, (b) a *child tree* is a tree with $(k+1)$ taxa that is generated from T_k by adding a new taxa into any one of its branches, (c) a *complete tree* is a tree with all n taxa in it, and (d) a *lower bound* for partial tree T_k^q is the minimum score of the complete tree, which is formed by adding all the remaining $(n-q)$ taxa to T_k^q. Our B&B approach

approach is as follows.

First, an upper bound U is specified by constructing a sub-optimal tree through heuristic algorithms, and Purdom et al.'s [24] lower bounding function is used to compute the lower bound of partial trees. In Purdom's lower bounding method, for a given partial tree, a *difference set* is computed for each site. A *difference set* is a set of characters with a cost associated to it. The elements in a *difference set* of a partial tree include all the characteristic features that do not occur in the partial tree but occur in a complete tree. The cost of a *difference set* is simply its cardinality. A lower bound for such a partial tree is the sum of the score of the partial tree and the cardinality of the *difference set.* Though there is no guarantee that using this method will improve the performance of the B&B algorithm, however, the number of partial trees that are decomposed is greatly reduced in many cases [24].

Next, each partial tree in the enumeration of the set of all $\Pi_{i=n}^{3}(2i-5)$ trees is considered in the following manner. Beginning with an initial three-taxa partial tree T_3, its first child tree T_4^1 is evaluated. If the lower bound of T_4^1 exceeds the upper bound U, then T_4^1 and all its child trees are eliminated from further consideration. If the lower bound does not exceed U, then each of the five child trees of T_4^1 is evaluated. If a partial tree is eliminated from further consideration, the search starts from another unconsidered partial tree in a depth-first manner. This is continued throughout the entire B&B search tree. During this process, when a complete tree is encountered at the last level of the B&B search tree, and its score $U' < U$, then U is reduced to U'.

Consider a dataset $D_{5,4}$ with 5 taxa and 4 sites shown in Table 2 and the corresponding

B&B search tree in Fig. 2. Let the specified upper bound for this dataset be U. An initial tree A with three randomly chosen taxa 1, 2, and 3 from $D_{5,4}$ is constructed. Let's call this the first level. Next taxa 4 from $D_{5,4}$ can be added to any of the three branches in A resulting in three trees B, C, and D at the second level. If Purdom's lower bound of B is greater than U, then B and all its child trees are eliminated. In such a case the next partial tree C is evaluated. To evaluate C, next taxa 5 is added to each branch in C resulting in five trees 1, 2, 3, 4, and 5 as shown in the figure. Since 1, 2, 3, 4, and 5 are complete trees, their score is compared with U for an improvement. For each of them, if their score $U' < U$, then $U = U'$. However, if Purdom's lower bound of B is less than U, then the next taxa 5 is added to all the branches in B resulting in five child trees. In this way, for any arbitrary dataset, the B&B search continues until all partial trees in the B&B search space are either eliminated or evaluated.

	S1	**S2**	**S3**	**S4**
1	A	A	C	D
2	A	B	D	D
3	A	B	E	I
4	A	B	F	I
5	A	C	G	I

Table 2: $D_{5,4}$ with 5 Taxa, and 4 Sites.

2.2 Determining Taxa Addition Order

In phylogenetic tree reconstruction, the order in which taxa are added to a partial tree greatly influences the performance of the B&B algorithm. While the order has no impact on correctness of the solution, it significantly affects the running time. During our experiments, we observed significant change in the number of partial trees decomposed by altering the

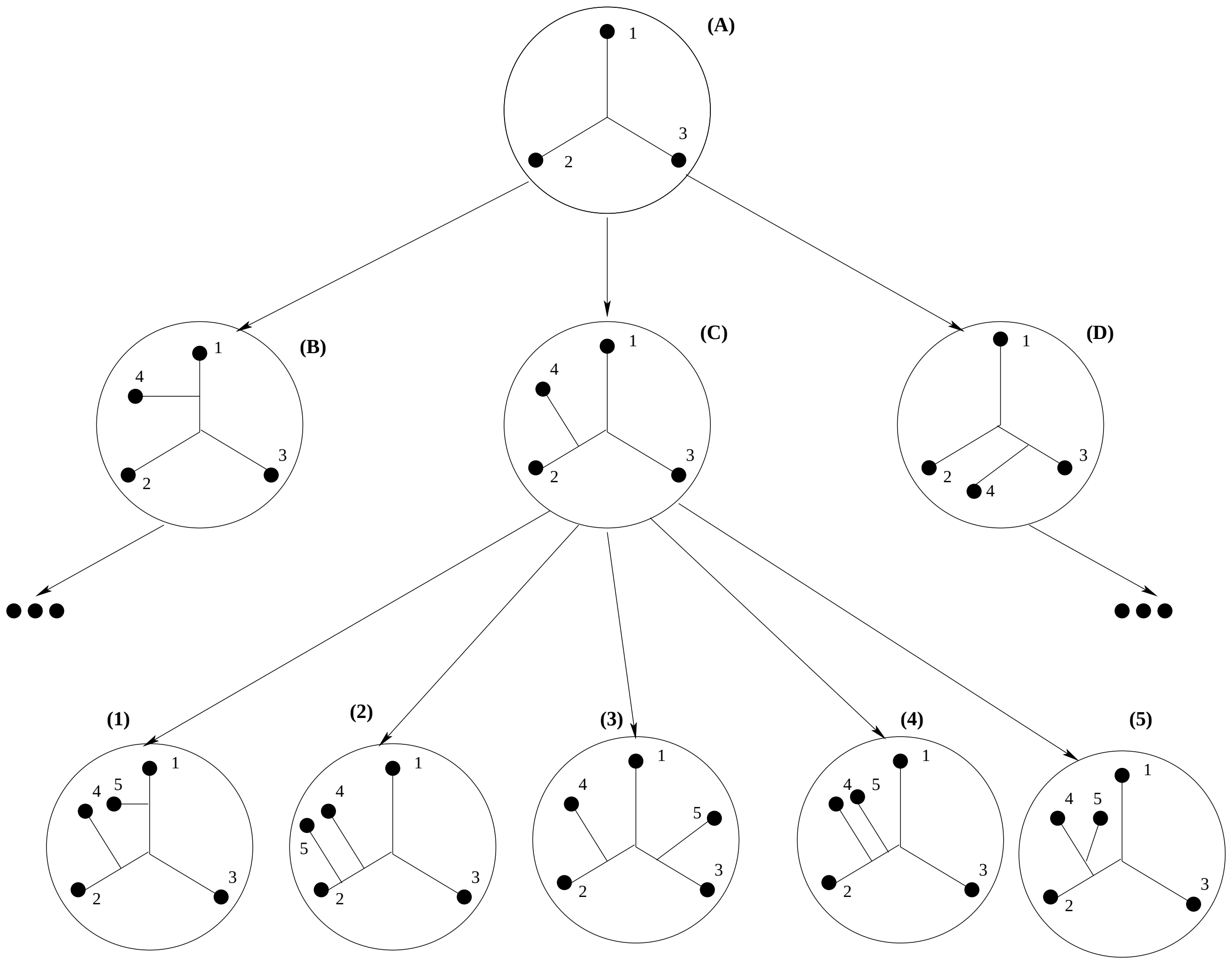

Figure 2: Phylogenetic Tree Construction for dataset $D_{5,4}$ shown in Table 2. 3^{rd} level contains 15 trees of which only five trees are shown.

taxa addition order. Nei and Kumar [22] propose an efficient heuristic for selecting a good addition order called *max_mini*. In max_mini, we start with an initial core tree T_p (where subscript p denotes parent) with three taxa, and decide the addition order of the remaining input sequences in the following way. Choose a taxa from the input dataset which is not yet added to T_p. Add the selected taxa in all possible branches j in T_p and compute the tree score L_i^j at each branch j. Note the branch j, where L_i^j is smallest (score L for i^{th} taxa at j^{th} branch). For all remaining taxa k, $k \neq i$, compute L_k^l in T_p (without taxa i added). If $L_k^l > L_i^j$, select k as next taxa to be added in branch l, else select i as next taxa to be added in branch j. Add selected taxa at the corresponding branch in T_p and remove it from the input dataset. Repeat until any taxa is remaining in the dataset. On an average, the max_mini technique reduced the execution times of the datasets in our test bed by a factor of up to 400.

2.3 Computing initial upper bound

Efficiency of any B&B algorithm is dependent on the tightness of its bounds. Ibaraki [14] discussed how the efficiency of B&B depends on tight upper and lower bounds. In ExactMP Solver, we employ a four-stage approach to compute an upper bound. Given a dataset and a taxa addition order, the first stage involves computing upper bound using Eck & Dayhoff's [6] greedy (EDG) algorithm. The EDG approach gives a tight upper bound in many cases. Consider a partial tree T_k in $D_{n,m}$. While the EDG approach searches for the best branch j for $(k+1)^{th}$ taxa in T_k to generate T_{k+1}, we suggest searching for the best branch j for all remaining taxa t, $k < t \leq n$ and select the (taxa, branch) pair which gives lowest tree score.

We call it the Improved EDG (IEDG) approach. Our results show that in all our datasets where EDG gives a loose upper bound, IEDG gives a tighter bound. This constitutes the second stage of computing the upper bound in ExactMP Solver. In third stage, we apply the Tree Bisection and Reconnection (TBR) [29] algorithm to the solutions of the IEDG approach. Although there is no guarantee of improvement, TBR usually results in a tree topology with lower tree score. In the fourth stage, we generate random sequences of taxa addition order and apply the EDG and IEDG approaches followed by TBR of their results. With this four-stage approach, we obtained tight upper bounds for all the datasets in our testbed. In our experiments, we found that with the application of our four stage technique to compute an initial upper bound for the datasets in our testbed, the running time reduced by a factor of 2.4.

2.4 Improved Fitch Kernel

Identifying and alleviating hot spots in a code is essential for best performance. Analyzing the phylogenetic reconstruction code, we found that over 85% of the time in our initial implementation was spent computing Fitch operations. A straightforward implementation of Fitch's operation for a single character requires at least one if-then-else branch. By considering processor-cache behavior, a more efficient method can be developed. Using a *Lookup_Table* for Fitch's operation is one such method. Since Fitch's operation is frequently referenced in phylogenetic reconstruction, for most of the time this table is likely to be placed inside cache. Hence, a Fitch's operation can often be performed in one cache lookup. However, it may be noted that if the size of table is of significant size then this method will not be suitable. In our experiments with a genome sequence consisting of four states, for five billion Fitch operations over three different architectures, on an average, the *Lookup_Table* scheme outperformed branching and *Multicharacter_Optimization* [21, 26] schemes by a factor of 17.4 and 10.3, respectively.

2.5 Removing Redundant Information

While there are typically many sites in a given dataset, not all contribute to the score of a phylogenetic tree. By ignoring the sites that do not contribute to the score of a phylogenetic tree, we may reduce computation. Fitch [9] made a basic classification of sequence sites as parsimony-informative, or parsimony non-informative depending on, whether a site contributes or not to the score of a phylogenetic tree. At a sequence site, if only one state appears, it is a singleton site. This is parsimony non-informative as the state of this site will never change for any tree topology. Such sites are ignored from parsimony analysis. However, if a site is non-singleton (site containing more than one state), that does not guarantee it to be parsimony informative. For a non-singleton site that contains at most one non-singleton state is again a parsimony non-informative site, because in all the possible tree topologies, the number of substitutions in its state remains the same. Hence, this can be ignored during the analysis. However, a constant factor is added in the end to the parsimony tree score that accounts for the constant increase in cost due to non-singleton states. The technique of site reordering and removing redundant information, on an average, reduced the running time by a factor of 4.1 on the datasets in our test bed.

For example, consider a dataset $D_{5,4}$, with 5 taxa and 4 sites (see Table 2). Site S1 contains one state **A** throughout. For all tree topologies, there will be no state change of this site and hence this is deemed to be a parsimony non-informative site. Site S2 contains two singleton states **A**, **C** and one non-singleton state **B**. For all tree topologies, the number of state changes from this site can be at most one (due to state **B**, other states are always constant). Hence, site S2 is a parsimony non-informative site contributing a constant increase in tree score. Site S3 with five singleton states is parsimony non-informative and hence tree topology can be assigned any of the five states at random. Site S4 with two non-singleton states is the sole parsimony informative site. While site S4 affects the phylogenetic tree, sites S1, S2, S3 do not. Hence, all the sites that are not parsimony-informative can be safely removed from MP analysis and adjusted later for the constant cost contributed by them.

A further improvement is to reorder the sites in a way that parsimony informative sites with highest number of non-singleton states (most parsimonious sites) are moved to the first contiguous locations. Since maximum change in tree score is due to the most parsimonious sites, this helps in computing the cost of a partial tree more effectively.

3 Parallel Implementation of ExactMP

Performance of the B&B algorithm is highly dependent on different factors, such as storage of open subtrees, choice of data structures, communication protocols, and choice of granularity. Benaïchouche et al. [4] provide an overview of ways to organize a storage structure as a priority queue. Usually, priority queues are represented by heaps where each parent has higher priority than its children. Various algorithms exist for heap management, for example Pairing Heap [10], skew-heap [32], D-heap [16], and Leftist heap [16]. Roucairol [27] showed shared, and distributed data structures. Biswas and Browne [5] present the first concurrent heap, latter improved by Rao and Kumar [25]. In Rao and Kumar's scheme, multiple processors concurrently access a binary-heap by acquiring locks on the nodes of the heap. Yan and Zhang [34] propose a Lock Bypassing algorithm that uses lock-on-demand and lock-bypassing techniques to minimize locking granularity. Jones [15] propose partial locking for skew heaps. [18] presents a good comparison of different data management schemes for concurrent PQs.

Several classic optimization problems, such as Vertex Cover, Graph Partitioning, and Quadratic Assignment have been solved in parallel using the B&B technique. Lüling and Monien [19] implemented a parallel B&B for the Vertex Cover problem and achieved a parallel speedup of 237.32 using a ring topology on a 256 processor system. Laursen [17] presents a parallel B&B for Quadratic Assignment, Graph Partitioning, and Weighted Vertex Cover problems and demonstrates a processor utilization factor of up to 0.954 through specialized communication protocols, such as Half-Surplus and On-Demand. However, the problem of phylogenetic tree reconstruction using exact maximum parsimony has not been implemented previously. Bader et al. in [3, 2] and Yan in [33] provide a framework of a parallel solver to solve this problem.

We parallelize the phylogenetic reconstruction algorithm in three simple steps: (a) perform the initial computations: preprocessing the input dataset, and computing initial global upper bound using heuristics as described in Section 2, (b) maintain a shared set (hereafter denoted as Σ) of open subtrees, and (c) let each processor synchronously select a subset of open subtrees from Σ, solve all the elements in the subset completely, and repeat step (c) until the set Σ is not empty. During this process, if a processor finds a tree whose score is lower than the current global upper bound, then it synchronously updates the global upper bound to the newly found value and continues. In the above three-step scheme, part (c) becomes challenging because different datasets of similar size behave very differently. Hence, the performance of the B&B algorithm is dependent on the difficulty level of a dataset. We loosely categorize the datasets into hard, moderate, and easy, depending on the number of subtrees that are decomposed while solving them. Typically, the number of subtrees decomposed in a hard dataset is larger by several orders than the number of subtrees decomposed in an easy dataset. Due to this reason, the time taken to solve a subtree of a hard dataset is also much higher than time taken to solve a subtree of similar size of an easy dataset.

Due to the difference in the number of subtrees decomposed, and the time taken to solve a subtree, between hard and easy datasets, a load balancing technique that is optimally tuned for a hard dataset may not work well for an easy dataset. This makes the design of a parallel solver for this problem that simultaneously achieves a load balance and high parallel speedup on all types of datasets is hard.

In ExactMP, in order to achieve a uniform load balance and good parallel speedup simultaneously, we designed and implemented two methods to parallelize this problem: List based and Queue based. In the List based method, the B&B search tree is dynamically decomposed to an appropriate level such that, on an average, each processor receives a desired number of subtrees. All subtrees in that level are maintained in a list according to Best-first (most promising subtree placed first) algorithm. For example, consider a dataset with n taxa to be run on p processors such that each processor receives at least k subtrees. An appropriate level up to which the search tree should be decomposed can be computed by recursively dividing pk by $(2i - 5)$ for all levels i ranging from 1 to $n - 2$. Once this is done, each processor locks the list, selects a subtree, unlocks the list, and solves the selected subproblem completely. This step is repeated until the list is empty. This method performs well on hard datasets because, in general subtrees in the B&B search space of a hard dataset takes long time to complete and hence, all processors finish up around the same time. However, in an easy dataset, this is not the case. Some subtrees may be solved very fast and some subtrees may go on for long time leaving some processors in idle state.

In the Queue based method, the B&B search tree is decomposed up to a certain level and all subtrees at that level are stored in a shared queue. A processor locks the queue, receives k subtrees based on the Best-first algorithm, and unlocks it. During this process, before unlocking the shared queue, each processor decomposes one or more subtrees from the set it received, and enqueues some subtrees back. This step ensures a uniform load distribution by guaranteeing that the shared queue holds some partial trees until the B&B search tree is not completely traversed.

In both methods, if a better score is found then it is globally updated. In the first method, the level to which the B&B search tree is decomposed becomes the tuning parameter, where as in the second method, the cardinality k of the set of subtrees that a processor receives from the shared queue becomes the tuning parameter. During our experiments, we found that a range of 5 up to 8 for the tuning parameter in List based method gave good results, and in the Queue based method, a value of 10 for the tuning parameter gave best results. Although both methods provide load balance equally well, parallel speedup is higher in the

Queue based method. Hence, we chose the Queue based method for our ExactMP Solver.

4 Experimental Results

Our experiments use an 8-way Western Scientific FusionA8, featuring 2.4 GHz AMD Opteron processors, each with 1 MB of cache memory, and 32 GB of shared memory. Our testbed consists of 18 datasets containing DNA sequences of varying sizes (from 12 to 27 taxa). We use three real datasets (a) Eukaryotes rDNA (27 taxa) (b) rbcl DNA (14 taxa), (c) Metazoan DNA (20 taxa), and fifteen randomly generated, hard, moderate, and easy, synthetic datasets.

Our experiments involved 8 processors for the above testbed. We achieved an average parallel speedup (measured w.r.t the best sequential time) of up to 6.9 on 8 processors using the List based method and 7.26 on 8 processors using the Queue based method. Fig. 3 shows the parallel speedup and load distribution of both methods for different datasets up to 8 processors. Highest speedup is seen on hard datasets due to the availability of subproblems. Load distribution is fairly uniform for all the datasets in both methods which is indicative of an efficient usage of processors. Uniform load distribution and high parallel speedup on hard datasets indicate that our parallelization techniques are highly efficient. However, uniform load distribution and a low parallel speedup in easy and real datasets point out that the sizes of the datasets are small enough to be solved on a parallel solver. This indicates that the ExactMP Solver can handle much larger instances of real and easy datasets efficiently.

Fig. 4 compares running times of ExactMP Solver and PAUP* for hard datasets. PAUP* is the state-of-the-art sequential code for constructing phylogenetic tree from genome sequence data using maximum parsimony. We used BranchAndBound operation in PAUP* with MulTree option turned on. This option allows the closest possible comparison between PAUP* and ExactMP Solver by instructing PAUP* to traverse the entire B&B search space. For all the datasets, ExactMP solver took less time than PAUP*.

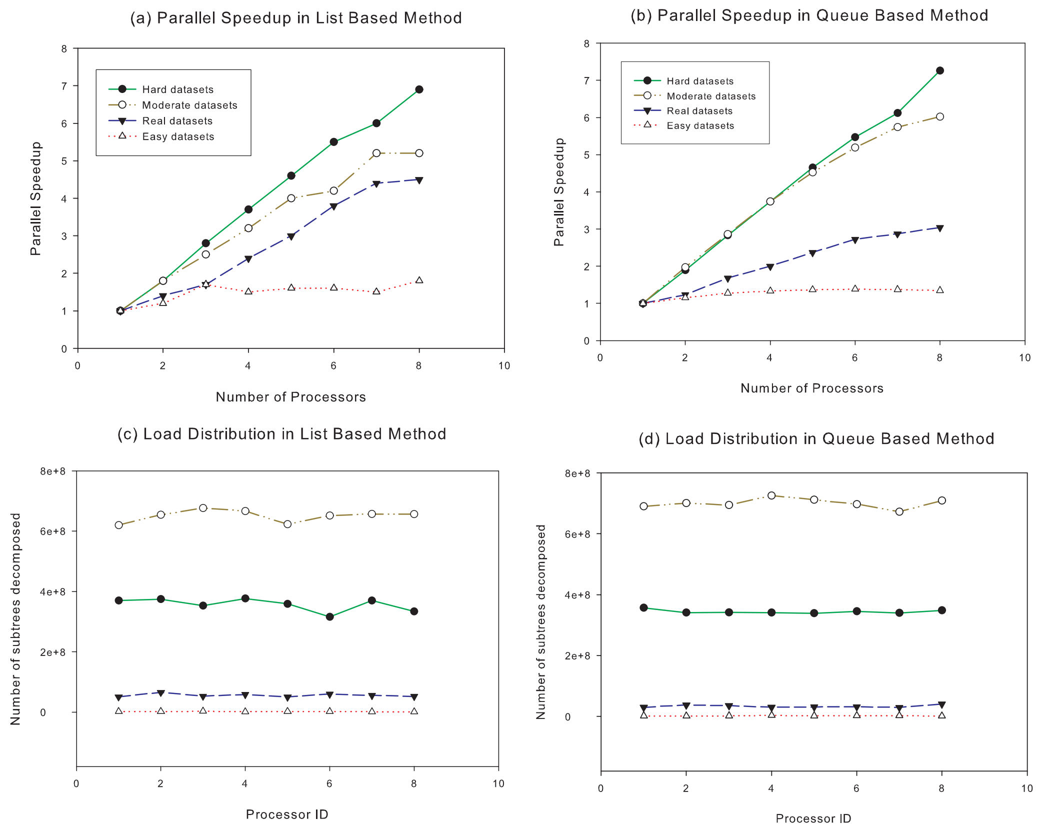

Figure 3: Average Parallel Speedup and Load Distribution for List and Queue based methods for easy, hard, moderate, and real datasets.

5 Conclusions and Future Work

We designed an efficient parallel solver for reconstructing exact phylogeny trees using maximum parsimony. Our new upper bounding methodology uses a combination of heuristic

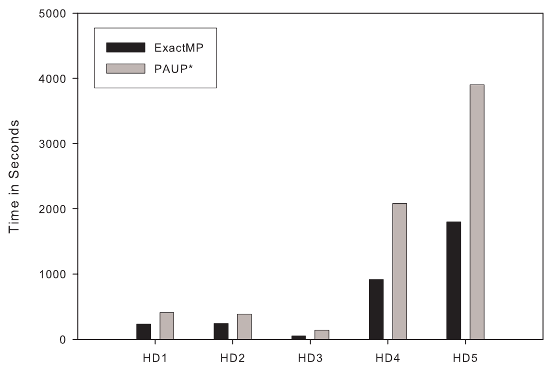

Figure 4: Comparison of ExactMP Solver using 8 processors with PAUP* on hard datasets.

and randomization techniques. We introduce a method to optimize the Fitch kernel that outperforms the well known multi-character optimization technique. Parallel speedups of up to 7.26 on 8 processors and a uniform load distribution pattern show that our parallelization techniques are efficient. The observation that the parallel speedup improves correspondingly with the difficulty level of a dataset indicates that ExactMP Solver is well suited for large instances of hard datasets and very large instances of other datasets. ExactMP Solver may also be used as a base method for the Disk Covering Method (DCM) [13] and to establish the accuracy of heuristic algorithms.

Our future research efforts in this area will be directed to making the ExactMP solver more general. Our goals in the future are to add optimization for protein sequences, schemes for missing character prediction, and incorporate Wagner, and Dollo parsimony strategies.

References

[1] Tree of Life: A distributed Internet project containing information about phylogeny and

biodiversity. Arizona State University. `http://tolweb.org/tree/phylogeny.html`.

[2] D.A. Bader, U. Roshan, and A. Stamatakis. Computational grand challenges in assembling the tree of life: Problems and solutions. The IEEE and ACM Supercomputing Conference 2005 (SC2005) Tutorial, Seattle, WA, November 13, 2005.

[3] D.A. Bader and M. Yan. High performance algorithms for phylogeny reconstruction with maximum parsimony. In S. Aluru, editor, *Handbook of Computational Molecular Biology*. Chapman & Hall, 2005.

[4] M. Benaïchouche, V. Cung, S. Dowaji, B.L. Cun, T. Mautor, and C. Roucairol. Building a parallel branch and bound library. In A. Ferreira and P. Pardalos, editors, *Solving Combinatorial Optimization Problem in Parallel: Methods and Techniques*, pages 201–231. Springer-Verlag, 1996.

[5] R. Biswas and J.C. Browne. Simultaneous update of priority structures. *Proc. Int'l Conf. on Parallel Processing*, pages 124–131, August 1987.

[6] R.V. Eck and M.O. Dayhoff. *Atlas of Protein Sequence and Structure*. National Biomedical Research Foundation, Silver Spring, MD, 1966.

[7] J. Farris. Methods for computing wagner trees. *Systematic Zoology*, 34:21–24, 1970.

[8] W.M. Fitch. Toward defining the course of evolution: Minimal change for a specific tree topology. *Systematic Zoology*, 20:406–416, 1971.

[9] W.M. Fitch. On the problem of discovering the most parsimonious tree. *The American Naturalist*, 111(978):223–257, 1977.

[10] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1:111–129, 1986.

[11] P.A. Goloboff. NONA. ver. 2., 1995.

[12] M.D. Hendy and D. Penny. Branch and bound algorithms to determine minimal evolutionary trees. *Mathematical Biosciences*, 59:277–290, 1982.

[13] D. Hudson, S. Nettles, L. Parida, T. Warnow, and S. Yooseph. The disk-covering method for tree reconstruction. *Proc. of Algorithms and Experiments (ALEX98)*, pages 62–75, February 1998.

[14] T. Ibaraki. Theoretical comparisons of search strategies in branch-and-bound algorithms. *Int'l Journal of Computer and Information Sciences*, 5(4):315–344, 1976.

[15] D.W. Jones. An empirical comparison of priority queues and event-set implementations. *Communications of the ACM*, 29:300–311, 1986.

[16] D.E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley Publishing Company, Reading, MA, 1973.

[17] P.S. Laursen. Experience with a synchronous parallel branch and bound algorithm. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 22:181–194, 1995.

[18] B. LeCun and C. Roucairol. Concurrent data structures for tree search algorithms. In A. Ferreira and J.D.P. Rolim, editors, *Parallel Algorithms for Irregular Problems: State of the Art*, pages 135–156. Kluwer Academic Publishers, 1995.

[19] R. Lüling and B. Monien. Load balancing for distributed branch and bound algorithms. In *Proc. 6th Int'l Parallel Processing Symposium*, pages 543–549, 1992.

[20] W.P. Maddison and D.R. Maddison. *MacClade: Analysis of phylogeny and character evolution.* Sinauer Associates, Sunderland, MA, 1992.

[21] A. Moilanen. Simulated evolutionary optimization and local search: Introduction and application to tree search. *Cladistics*, 17:512–525, 2001.

[22] M. Nei and S. Kumar. *Molecular Evolution and Phylogenetics.* Oxford University Press, Oxford, UK, 2000.

[23] K.C. Nixon. The parsimony ratchet, a new method for rapid parsimony analysis. *Cladistics*, 15:407–414, 1999.

[24] P.W. Purdom, Jr., P.G. Bradford, K. Tamura, and S. Kumar. Single column discrepancy and dynamic max-mini optimization for quickly finding the most parsimonious evolutionary trees. *Bioinfomatics*, 2(16):140–151, 2000.

[25] V.N. Rao and V. Kumar. Concurrent access of priority queues. *IEEE Transactions on Computers*, C-37:1657–1665, 1988.

[26] F. Ronquist. Fast Fitch-parsimony algorithms for large data sets. *Cladistics*, 14(4):387–400, 1998.

[27] C. Roucairol. On irregular data structures and asynchronous parallel branch and bound algorithms. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 22:323–336, 1995.

[28] D.L. Swofford. Paup: Phylogenetic analysis using parsimony, version 3.1. program and documentation, 1993.

[29] D.L. Swofford and D.P. Begle. *PAUP: Phylogenetic analysis using parsimony.* Sinauer Associates, Sunderland, MA, 1993.

[30] D.L. Swofford and W.P. Maddison. Reconstructing ancestral character states using wagner parsimony. *Math. Biosci*, pages 199–299, 1987.

[31] D.L. Swofford, G.J. Olsen, P.J. Waddell, and D.M. Hillis. Phylogenetic inference. In D.M. Hillis, C. Moritz, and B.K. Mable, editors, *Molecular Systematics*, pages 407–514. Sinauer, Sunderland, MA, 1996.

[32] R. Tarjan and D. Sleator. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[33] Mi Yan. *High Performance Algorithms for Phylogeny Reconstruction with Maximum Parsimony*. PhD thesis, University of New Mexico, Albuquerque, Department of Electrical & Computer Engineering, May 2004.

[34] Y. Yan and X. Zhang. Lock bypassing: An efficient algorithm for concurrently accessing priority heaps. *ACM J. Experimental Algorithmics*, 3(3), 1998. `www.jea.acm.org/1998/YanLock/`.