Parallelizing the Phylogeny Problem

Jeff A. Jones

December 14, 1994

1 Introduction

Determining the evolutionary history of species is fundamental to molecular biology. The problem of determining the evolutionary history for a set of species is known as the phylogeny problem. Evolutionary history is typically represented by a tree, with the root being the oldest common ancestor and the children of a node being the species that evolved directly from that node. A path from the root to a species shows the evolutionary path of that species. The phylogenetic tree shows relationships between species and is an important tool in the branch of biology known as systematics. It is valuable in itself, but also provides clues about migration patterns, climate changes, the formation of the earth, and many other mysteries of the diversity of life.

Techniques for sequencing proteins and nucleic acids provide the foundation for the phylogeny problem. Molecular sequences contain precise, quantitative information present in a wide variety of organisms. Solving the phylogeny problem means deriving, from a given set of molecular sequences, an evolutionary history in the form of a phylogenetic tree.

Unfortunately, deriving phylogenetic trees is often prohibitively expensive. We used careful software design and parallel processing to solve the phylogeny problem quickly.

Many methods for solving the phylogeny problem have been explored, including parsimony, compatibility, maximum likelihood, and distance matrix methods [3]. We focus on the method known as character compatibility [7], which we describe formally in Section 2. The character compatibility method searches a large space, so we have developed a number of heuristics for bounding the search. We demonstrate that the heuristics have a considerable impact on performance.

Solving the character compatibility problem requires, as a subproblem, the solution of the *perfect phylogeny problem*, a restriction of the general phylogeny problem. Character compatibility solves the general phylogeny problem by combining the solutions of several perfect phylogeny problems. Our character compatibility implementation relies on an algorithm due to Agarwala and Fernández-Baca [1] to solve the perfect phylogeny problem. We give a new, simpler description and proof of correctness of this algorithm in Section 3.

Deriving an efficient implementation from an efficient algorithm requires care. Maximum performance requires appropriate data structures and use of heuristics based on expected inputs. In Section 4, we describe our sequential implementation, including the important data structures and our techniques for improving performance of both the character compatibility problem and the perfect phylogeny problem. We present timing results for our implementation.

Finally, we describe our parallel implementation on a distributed memory multiprocessor in Section 5. The important decisions in the implementation concern the sources of parallelism and the amount of data sharing. Multiple levels of parallelism are available, but we use only one. We also limit the amount of data sharing to simplify the implementation. We conclude the section by suggesting improvements to the implementation.

2 The Character Compatibility Problem

Phylogenetic trees are constructed by considering the *characters* exhibited by the species. The characters can be skeletal structures, coloring, or other physical characteristics. More often, the characters are elements of molecular sequences. We represent a species u with a vector of character values, $u[1], \ldots, u[c_{max}]$, where c_{max} is the number of characters to be considered. In the case of molecular sequences, each element of this vector is a nucleotide or amino acid.

A character is *compatible* with a phylogenetic tree if no value for that character arises more than once along any path in the tree. For example, if a species loses some trait, such as opposable thumbs, it cannot regain it. A solution of the character compatibility problem is a phylogenetic tree with the maximum number of compatible characters.

A *perfect phylogenetic tree* for a set of species and a given set of characters for those species is a phylogenetic tree compatible with all the characters. Note that the phylogeny problem does not find roots for the trees. Perfect phylogenetic trees only indicate the relative evolutionary distances between species. The root must be calculated by considering other factors, such as the ages of the species. Formally, we define a perfect phylogenetic tree, or perfect phylogeny for short, for a fixed set of characters, as follows:

Definition 1 *Given a set of species S and a set of characters C, an undirected tree T is a* perfect phylogenetic tree *for S with C if*

1. *$S \subseteq V(T)$, the vertices of T*
2. *Every leaf in T is in S*
3. *For all paths $u_0, \ldots, u_k$, for all characters $c \in C$, if $u_0[c] = u_k[c]$, then for all i, $0 \leq i \leq k$, $u_i[c] = u_0[c]$*

If a perfect phylogeny exists for a set of species with a particular set of characters, we say that the set of characters is *compatible*.

To illustrate the definition, consider Figure 1, in which we consider a set of $n = 3$ species, $\{u, v, w\}$, with three characters each ($c_{max} = 3$). Each character takes on one of up to 4 possible values. Tree a is not a perfect phylogeny because it violates condition 3 in the definition: $u[2] = w[2]$, but $v[2] \neq u[2]$. Tree b satisfies all three conditions, as does tree c. Notice that the perfect phylogeny for a given set of species may not be unique.

Tree c contains the species $[1, 1, 3]$, which was not a member of the original set. The tree is still a perfect phylogeny, however, because all of the leaves appear in the original set. In fact, some sets of species have no perfect phylogenies containing only members of the set, indicating the existence of an unknown intermediate species, or "missing link," in the evolutionary history of the set.

To make the example more concrete, consider u and w to be species with opposable thumbs, and v to be a species without this trait. Tree a is not a valid phylogeny because, in some evolutionary path, opposable thumbs were lost and then reappeared.

Ideally, we would like to find a perfect phylogeny for our original set of species with the original set of characters. That is, we hope that the entire set of characters is compatible. Unfortunately, this is not true for many sets of species. Table 1 shows an example. Even

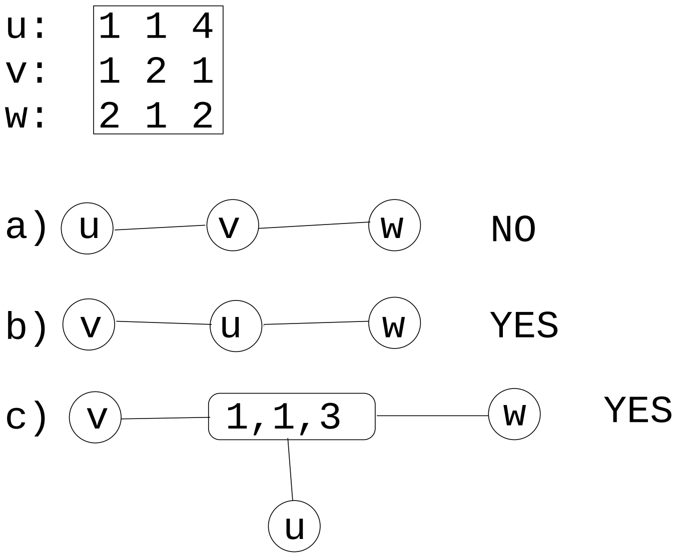

Figure 1: Examples of perfect phylogenetic trees.

u	1	1
v	1	2
w	2	1
x	2	2

Table 1: Set that does not have a perfect phylogeny.

adding new internal vertices does not produce a perfect phylogeny. Because a character set may be incompatible, the character compatibility method searches for the largest compatible subset. If the subset is large, the corresponding perfect phylogeny will be a good estimate of the evolutionary history of the species.

Solving the character compatibility problem requires determining whether or not each subset of characters is compatible. We use the algorithm of Agarwala and Fernández-Baca [1], which we describe in Section 3, to determine compatibility. Examining all possible subsets (an exponential number) in turn is extremely time-consuming, however. Fortunately, we can reduce the number of subproblems examined with the following lemma.

Lemma 1 *Let S be a set of species, and let C_1 and C_2 be subsets of the characters of the species in S, such that $C_1 \subseteq C_2$. If C_2 is compatible, C_1 is compatible.*

Proof. If C_2 is compatible, there exists a perfect phylogeny T for S that is compatible with every member of C_2. Because any member of C_1 is a member of C_2, it must be compatible with T, and therefore C_1 is compatible.
□

Once we find a compatible set, we know that all subsets of that set are also compatible. Similarly, if we find an incompatible set, we can exclude all supersets of that set from consideration. We can consider the subsets as being partially ordered by the subset relation, whose lattice is shown in Figure 2, for the 3 character case. Then, the goal of the character compatibility problem is to find a frontier of compatible subsets in this lattice. Figure 3 shows the

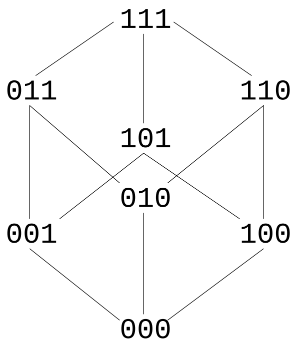

Figure 2: Lattice for the 3 character case.

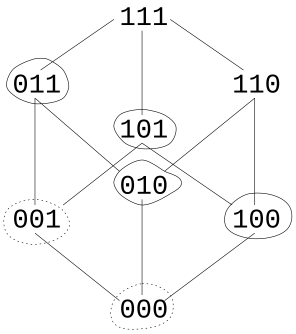

Figure 3: Example frontier.

result for the species in Table 2. Compatible subsets are circled in dashed lines, and members of the compatibility frontier are circled in solid lines. Section 4 will discuss the performance ramifications of this structure.

3 The Perfect Phylogeny Problem

We focus now on solving the perfect phylogeny problem. We fix the set of characters and derive an algorithm that decides if a perfect phylogeny compatible with that set exists.

The general form of the problem is NP-complete [2]. Possible approaches to obtaining a polynomial-time algorithm are fixing the maximum number of characters and fixing the maximum number of possible values per character to a constant, r_{max}. Using the first approach, it has been shown that the problem may be solved in time $O(n^{m+1})$, where n is the number of species and m is the number of characters [8]. We describe an algorithm using the second approach, due to Agarwala and Fernández-Baca, which achieves a time bound of

u	1	1	1
v	1	2	1
w	2	1	1
x	2	2	1

Table 2: Another example set of species.

$O(2^{2r_{max}}(nm^3 + m^4))$. This approach promises to be more practical, because typical values of r_{max} are 4 for nucleotide sequences and 20 for proteins, but the number of characters in a sequence can be in the thousands.

The algorithm divides the set of species into smaller sets, and proceeds recursively on the smaller sets. The two methods for dividing the set are vertex decomposition and edge decomposition. Vertex decomposition divides the species into two subsets so that we can build a perfect phylogeny with some member of the original set of species as an internal vertex. The internal vertex will be on every path between members of different sets, so condition 3 of Definition 1 will restrict the possible candidates. For some sets of species, however, no species satisfies these requirements, and we resort to edge decomposition, which corresponds to adding a new species.

3.1 Vertex Decomposition

To describe vertex decomposition, we first introduce the notions of common character values and the common vector between two sets of species.

Definition 2 *For two sets of species S_1 and S_2, and a character c, if there exists $u_1 \in S_1$ and $u_2 \in S_2$ such that $u_1[c] = u_2[c]$, then $u_1[c] = u_2[c]$ is a* common character value between S_1 and S_2 for c.

Definition 3 *If there is at most one common character value between S_1 and S_2 for each character c, then the* common vector *for S_1 and S_2 is a vector $\mathrm{cv}(S_1, S_2)$ such that*

$$\mathrm{cv}(S_1, S_2)[c] = \begin{cases} r_c & \textit{if } r_c \textit{ is the common character value for } c \\ \text{unforced} & \textit{if there is no common character value for } c \end{cases}$$

If there is more than one common character value between S_1 and S_2 for some character, then we say the common vector is not defined.

In this definition, "unforced" is a new character value that requires special treatment:

Definition 4 *We say two vectors u and v are* similar *if, for all c, either $u[c] = v[c]$, $u[c] =$ unforced, or $v[c] =$ unforced.*

If a set S contains species that have unforced character values, then we say a tree T is a perfect phylogeny for S if and only if T is a perfect phylogeny for some set S', where $S \subseteq S'$ and S' contains, for every species in S, a species which has a similar, fully-forced character vector. Such sets of species arise because of edge decomposition, described in Section 3.2.

Consider all pairs (S_1, S_2), such that $S = S_1 \cup S_2$ and $S_1 \cap S_2 = \emptyset$. If the common vector $\mathrm{cv}(S_1, S_2)$ is defined, we call the pair (S_1, S_2) a *split*. If, for one of these splits, $\mathrm{cv}(S_1, S_2)$ is similar to some species $u \in S$, the split is called a *vertex decomposition*. Intuitively, finding a vertex decomposition corresponds to finding a member of the set of species that can be an internal vertex of a perfect phylogeny for the set. The restriction that u be compatible with the common vector between the two sets derives from the fact that in the perfect phylogeny, u will be on a path between any members of opposing sets. The following lemma formalizes this statement, and shows how a vertex decomposition can be used to find a perfect phylogeny.

Lemma 2 *Assume that a set of species S has a vertex decomposition (S_1, S_2), where $u \in S$ is the species which is similar to $\mathrm{cv}(S_1, S_2)$. Then, S has a perfect phylogeny if and only if both $S_1 \cup \{u\}$ and $S_2 \cup \{u\}$ have perfect phylogenies.*

u:	1	1
v:	2	3
w:	2	1
x:	4	3
y:	2	4

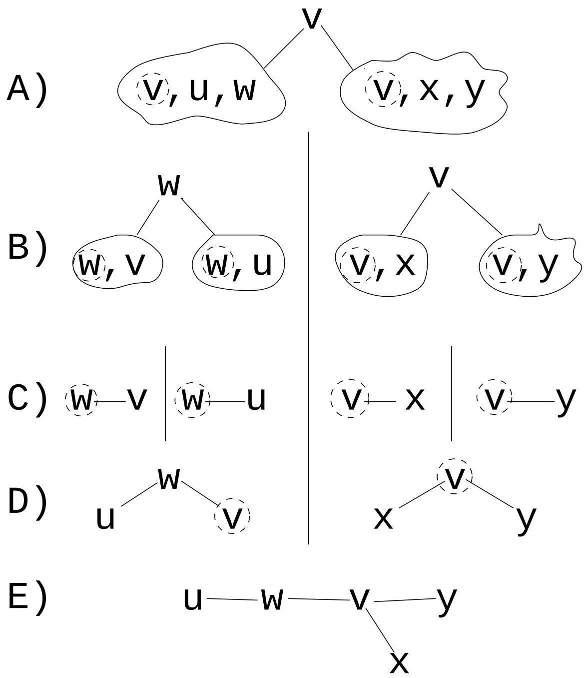

Figure 4: Vertex decomposition example.

Proof. If perfect phylogenies, T_1 and T_2, exist for $S_1 \cup \{u\}$ and $S_2 \cup \{u\}$, respectively, we can construct a perfect phylogeny for S by simply merging the nodes for u in T_1 and T_2. That is, we produce a new tree T which contains all the vertices and edges of T_1 and T_2. To guarantee that T is a perfect phylogeny, however, we must first modify T_1 and T_2 to ensure that $\mathrm{cv}(V(T_1), V(T_2))$ is similar to $\mathrm{cv}(S_1, S_2)$. Only vertices that are not in the original set of species and that have character values that are not "forced" by condition 3 of Definition 1 can violate this condition. We can modify these character values to be equal to that of some neighboring vertex that is a member of S, however, which solves the problem.

If S has a perfect phylogeny T, we can produce a perfect phylogeny for any subset S' of S, by removing leaves of T that are not members of S'. Therefore, if either $S_1 \cup \{u\}$ or $S_2 \cup \{u\}$ does not have a perfect phylogeny, neither does S.
□

To construct a perfect phylogeny for S, we look for a vertex decomposition and proceed recursively on the subsets. Finding a perfect phylogeny for a set with 1 or 2 elements is trivial, so the recursion terminates. If at any point we fail to find a phylogeny for a subset, we know that none exists for S.

u:	2	1	1
v:	1	1	2
w:	1	2	1

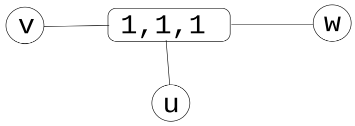

Figure 5: A perfect phylogeny for a set with no vertex decompositions.

Figure 4 shows the process of finding a perfect phylogeny for a set of five species. In step A, the common vector between $S_1 = \{v, u, w\}$ and $S_2 = \{x, y\}$ is $[2, 3]$, which is similar to v. We look for perfect phylogenies for $\{v, u, w\}$ and $\{v, x, y\}$. In the figure, the dashed circles in each step indicate the species used to connect the two subtrees for that step.

In step B, we decompose on w and v in their respective sets, leaving a collection of subsets of size 2, which have obvious perfect phylogenies. A construction for a perfect phylogeny for any set of three species also exists, but we avoid its use here for simplicity. To complete the process, we connect the subtrees into successively larger trees by connecting the appropriate vertices.

Unfortunately, vertex decomposition does not suffice for all sets of species. For example, Figure 5 shows a set of species that has no vertex decomposition, but has a perfect phylogeny. Notice that, as in tree c of Figure 1, we have added a new vertex with character vector $[1, 1, 1]$. As noted above, a vertex decomposition requires a member of the original set of species that can be an internal vertex of the perfect phylogeny. The set in Figure 1, however, has no such species, which explains the necessity for the added species.

To account for the necessity of adding vertices, we extend the perfect phylogeny procedure. When the search for a vertex decomposition fails, we search for an *edge decomposition*.

3.2 Edge Decomposition

For vertex decomposition, we make use of splits that have common vectors that are similar to one of the species. But, if no such similar species is present, the split may still be useful if it is a *c-split*:

Definition 5 *A* c-split *is a split (S_1, S_2) such that for at least one character c, there is no common character value between S_1 and S_2.*

To understand the definition of a c-split, assume that the set of species S has a perfect phylogeny T. Without loss of generality, assume the vertices of T are distinct (we could simply merge identical nodes). For any edge $\{u_1, u_2\}$ of T, consider T_1 and T_2, the subtrees containing at u_1 and u_2, respectively. Let S_1 and S_2 be the members of S that are vertices of T_1 and T_2, respectively. Because T is a perfect phylogeny, (S_1, S_2) can be shown to be a c-split, as follows. The common vector between S_1 and S_2 is defined because there is only one edge, namely $\{u_1, u_2\}$, connecting S_1 and S_2, so if there were more than one common character value between S_1 and S_2, T could not be a perfect phylogeny. And, because u_1 and u_2 are distinct, $u_1[c] \neq u_2[c]$ for some character c, so there can be no common character value between S_1 and S_2 for c. Therefore, if a set S has a perfect phylogeny, it must have a c-split for every edge in the tree.

This fact suggests a possible algorithm for finding a perfect phylogeny. For all c-splits (S_1, S_2) of S, we could attempt to find perfect phylogenies for S_1 and S_2. If any c-split has two such perfect phylogenies, we can build a perfect phylogeny for S. If no c-split does, then no perfect phylogeny exists for S by the reasoning above.

Unfortunately, the naïve algorithm relying on a recursive search takes time exponential in the number of species. The algorithm is unacceptable because the number of species in a problem instance tends to be large. Nevertheless, with some modification, we can use memoization to reduce the running time to be polynomial in the number of species.

We can build all perfect phylogenies for S from perfect phylogenies for subsets with a particular form.

Definition 6 *Let S be the original set of species. For any subset S_1 of S, define $\bar{S_1} = S - S_1$.*

Definition 7 *For an original set of species S, if $(S_1, \bar{S_1})$ is a split and the set $S_1 \cup \{\mathrm{cv}(S_1, \bar{S_1})\}$ has a perfect phylogeny, we call this tree a* subphylogeny for S_1 in S.

The idea behind this definition is that the vertex of the subphylogeny corresponding to $\mathrm{cv}(S_1, \bar{S_1})$ can be used to connect to a perfect phylogeny for the rest of the set. The lemma below explains the details. Intuitively, it states that we can build a perfect phylogeny for any subset S' such that $(S', \bar{S'})$ is a split by finding subphylogenies only for subsets S_1 where $(S_1, \bar{S_1})$ is a c-split.

Lemma 3 *For any split $(S', \bar{S'})$ of the original set of species S, S' has a subphylogeny if and only if there exists a c-split (S_1, S_2) of S' such that:*

1. *$(S_1, \bar{S_1})$ is a c-split*
2. *$\mathrm{cv}(S_1, S_2)$ is similar to $\mathrm{cv}(S', \bar{S'})$*
3. *S_1 has a subphylogeny*
4. *S_2 has a subphylogeny*

Proof. First, assume we have such a c-split. Let T_1 and T_2 be the subphylogenies for S_1 and S_2, respectively, and let cv_1 and cv_2 be the vertices in the subphylogenies corresponding to $\mathrm{cv}(S_1, \bar{S_1})$ and $\mathrm{cv}(S_2, \bar{S_2})$, respectively. Note that $\mathrm{cv}(S_1, \bar{S_1})$ must be similar to $\mathrm{cv}(S_2, \bar{S_2})$, and both of these must be similar to $\mathrm{cv}(S', \bar{S'})$. This follows, after tedious case analysis, from the fact that $\mathrm{cv}(S_1, S_2)$ is similar to $\mathrm{cv}(S', \bar{S'})$. We can construct a perfect phylogeny as follows. Let cv be a vertex such that $\mathrm{cv}[c] = \mathrm{cv}(S', \bar{S'})[c]$ if $\mathrm{cv}(S', \bar{S'})[c]$ is forced, otherwise $\mathrm{cv}[c] = \mathrm{cv}(S_1, S_2)[c]$ if $\mathrm{cv}(S_1, S_2)[c]$ is forced, and otherwise $\mathrm{cv}[c] = \mathrm{cv}_1[c]$. Now, connect cv_1 and cv_2 to cv, as shown in Figure 6.

This tree is a perfect phylogeny. Otherwise, there must exist nodes $a, b \in V(T)$ such that, for some c, $a[c] = b[c]$ and there exists a node d on the path between a and b with $d[c] \neq a[c]$. Clearly, we cannot have both a and b in $V(T_1)$, because then T_1 would not be a perfect phylogeny. Similarly, we cannot have both a and b in $V(T_2)$. Therefore, we have three cases: $a \in V(T_1)$ and $b \in V(T_2)$; $a \in V(T_1)$ and $b = \mathrm{cv}$; and $a \in V(T_2)$ and $b = \mathrm{cv}$. In the first case, we must have $a[c] = b[c] = \mathrm{cv}_1[c] = \mathrm{cv}_2[c] = \mathrm{cv}[c]$, because $\mathrm{cv}(S_1, S_2)[c] = a[c]$, and $\mathrm{cv}(S_1, S_2)$ is similar to cv. Here, we assume that if a vertex in $V(T_1)$ or $V(T_2)$ is not a member of the original set of species, then each of its character values is the same as the character value of some species in the original set. Note that cv satisfies this criterion. So, all nodes d on the path are either between a and cv_1, in which case $d[c] = a[c]$, because T_1 is a perfect phylogeny, between b and cv_2, in which case $d[c] = b[c]$ because T_2 is a perfect phylogeny, or are cv, in which case $d[c] = a[c]$ because $\mathrm{cv}[c] = a[c]$. The remaining two cases are similar.

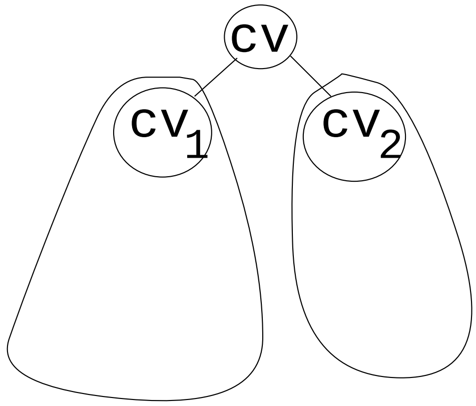

Figure 6: Edge decomposition example.

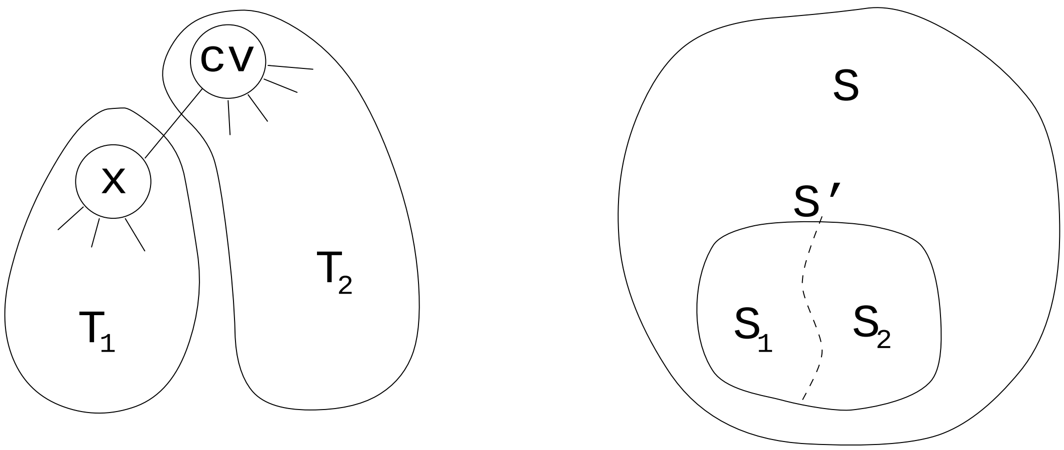

Figure 7: The situation

Now, to prove the other direction, assume that we have a perfect phylogeny T for $S' \cup \mathrm{cv}(S', \bar{S}')$. Assume, without loss of generality, that all nodes of T are distinct. Let cv be the vertex of T corresponding to $\mathrm{cv}(S', \bar{S}')$, and let x be any neighbor of cv. Remove the edge $\{x, \mathrm{cv}\}$, and let T_1 and T_2 be the two connected components. Assume $x \in V(T_1)$, and let $S_1 = V(T_1) \cap S'$ and $S_2 = V(T_2) \cap S'$. Clearly, (S_1, S_2) is a c-split, because it corresponds to the deletion of the edge $\{x, \mathrm{cv}\}$. Figure 7 depicts the situation. We now show that each of the four conditions hold.

1. To show that $(S_1, \bar{S}_1)$ is a c-split, we need to show that the common vector between S_1 and $\bar{S}_1$ is defined and that it is unforced in at least one character. The common vector is defined because if there is a common character value between S_1 and $\bar{S}'$, cv has that value for that character, because it is similar to $\mathrm{cv}(S', \bar{S}')$. Therefore, if the common vector is defined, it must also be similar to $\mathrm{cv}(S', \bar{S}')$. In effect, cv represents the elements of $\bar{S}'$ in T. Also, if there is a common character value between S_1 and S_2, cv has that value for that character because in T, cv is between any species in S_1 and any species in S_2, and T is a perfect phylogeny. In other words, if the common vector is defined, it must be similar to $\mathrm{cv}(S_1, S_2)$. The result is that there cannot be more than one common character value

between S_1 and $\bar{S}_1$, because otherwise cv would have two values for the same character. Therefore, the common vector $\mathrm{cv}(S_1, \bar{S}_1)$ is defined.

Now we will show that this common vector is unforced in at least one character by contradiction. Assume that $\mathrm{cv}(S_1, \bar{S}_1)$ is forced for all characters. As shown above, this vector is similar to $\mathrm{cv}(S', \bar{S}')$, which is similar to cv. Because $\mathrm{cv}(S_1, \bar{S}_1)$ is fully forced, it is therefore equal to cv. However, in T, x lies between all species in S_1 and cv. T is a perfect phylogeny, so x must be equal to cv also. This contradicts our assumption that the nodes of T are distinct. Therefore, the common vector is unforced in at least one character.

2. Next, we must show that $\mathrm{cv}(S_1, S_2)$ is similar to $\mathrm{cv}(S', \bar{S}')$. This follows easily from the observation that any path from a member of S_1 to a member of S_2 in T passes through cv. Therefore, $\mathrm{cv}(S_1, S_2)$ is similar to cv, which is fully defined and similar to $\mathrm{cv}(S', \bar{S}')$, so $\mathrm{cv}(S_1, S_2)$ is similar to $\mathrm{cv}(S', \bar{S}')$.

3. T_1 is clearly a perfect phylogeny for $S_1 \cup \{x\}$, so to show that S_1 has a subphylogeny, we only need to show that x is similar to $\mathrm{cv}(S_1, \bar{S}_1)$. We know, from above, that both $\mathrm{cv}(S', \bar{S}')$ and $\mathrm{cv}(S_1, S_2)$ are similar to cv, which implies that $\mathrm{cv}(S_1, \bar{S}_1)$ is similar to cv. If $\mathrm{cv}(S_1, \bar{S}_1)$ is forced for some character c, some species $u_c \in S_1$ exists such that $u_c[c] = \mathrm{cv}[c]$. Therefore, because x lies between u_c and cv in T, $x[c] = \mathrm{cv}[c] = \mathrm{cv}(S_1, \bar{S}_1)[c]$. x is equal to $\mathrm{cv}(S_1, \bar{S}_1)$ wherever $\mathrm{cv}(S_1, \bar{S}_1)$ is forced, which implies that x is similar to $\mathrm{cv}(S_1, \bar{S}_1)$.

4. Showing that S_2 has a subphylogeny is similar to the above argument: T_2 is a perfect phylogeny for $S_2 \cup \{\mathrm{cv}\}$, and cv is similar to $\mathrm{cv}(S_2, \bar{S}_2)$.

□

By the preceding lemma, the procedure Subphylogeny in Figure 8 determines if a perfect phylogeny exists for a subset S', where $(S', \bar{S}')$ is a split. Subphylogeny searches for a c-split of the set that satisfies each of the four conditions in the lemma. Notice that, for efficiency, the procedure calls itself only when all other conditions are met. Note that $\oplus$ denotes the merging of similar vectors, defined as follows:

$$(a \oplus b)[c] = \begin{cases} a[c] & \text{if } a[c] \text{ is forced} \\ b[c] & \text{if } b[c] \text{ is forced} \\ \text{unforced} & \text{otherwise} \end{cases}$$

Showing that $\mathrm{cv}(S_1, \bar{S}_1) = \mathrm{cv}(S_1, S_2) \oplus \mathrm{cv}(S', \bar{S}')$ is not difficult.

The more efficient procedure shown in Figure 9 reuses previously computed information about the existence of subphylogenies. Whenever Subphylogeny2 determines the existence or nonexistence of a subphylogeny for a subset, it places that result in a store of results. It checks the store before working further on a subset.

Because Subphylogeny2 never examines a set S', where $(S', \bar{S}')$ is a c-split, more than once, we can bound its running time by estimating the number of c-splits of a set of species. Specifically, there are at most $m2^{r_{max}-1}$ c-splits, where m is the number of characters and r_{max} is the number of values per character. The reason is that there is at most one c-split for each subset of the values of each of m characters. Each character has $2^{r_{max}}$ subsets of values, and half of these are duplicates. Therefore, as long as Subphylogeny2 requires only a polynomial amount of work to find each subphylogeny, it will be polynomial in the number of species and the number of characters per species. It will be exponential in the number of values per

```
Subphylogeny( S', cv )
{
    if |S'| = 1
        return TRUE
    else
        for each c-split (S_1, S_2) of S' {
            if (cv(S_1, S_2) is similar to cv) and either (S_1, S̄_1) or (S_2, S̄_2) is a c-split)
                if Subphylogeny( S_1, cv(S_1, S_2) ⊕ cv ) and Subphylogeny( S_2, cv(S_1, S_2) ⊕ cv )
                    return TRUE
        }
        return FALSE
}
```

Figure 8: A simple procedure for the perfect phylogeny problem.

```
Subphylogeny2( S', cv )
{
    if |S'| = 1
        return TRUE
    else if a result for S' is in the store
        return the previously computed result
    else
        for each c-split (S_1, S_2) of S' {
            if cv(S_1, S_2) is similar to cv and either (S_1, S̄_1) or (S_2, S̄_2) is a c-split
                if Subphylogeny2( S_1, cv(S_1, S_2) ⊕ cv ) and Subphylogeny2( S_2, cv(S_1, S_2) ⊕ cv )
                    record TRUE for S' in the store
                    return TRUE
        }
        record FALSE for S' in the store
        return FALSE
}
```

Figure 9: An improved algorithm for the perfect phylogeny problem.

character, but this weakness is acceptable because the number of values per character is small for molecular sequences, as mentioned above.

4 Sequential Implementation

Transforming an algorithm into an efficient implementation requires careful design. In this section, we describe our sequential implementation.

4.1 Exploring the Space of Subsets

Recall that the character compatibility problem is the problem of finding the largest subset of characters that has a perfect phylogeny. We must explore the space of all subsets of the original set of characters, using the perfect phylogeny procedure to decide success or failure for each subset. Fortunately, we can eliminate many subsets from consideration. Lemma 1 is our tool for limiting the character compatibility search space.

The conceptually simplest technique for attacking character compatibility is to enumerate all subsets of the original set of characters and step through them one by one, calling the procedure from Section 3 each time. This algorithm is horribly inefficient, however, because it uses none of the information obtained from the solution of other subsets.

As a first refinement, we maintain a store of results from previously examined subsets. We step through the subsets as before, but before calling the perfect phylogeny procedure for a subset S', we search the store for any supersets of S' already found compatible and for any subsets of S' already found incompatible. We refer to these two cases as successes and failures, respectively. In both cases, we avoid resorting to the perfect phylogeny procedure.

This improved algorithm is a marked improvement, but we can eliminate much more redundant work. To estimate the magnitude of the problem, we note that a 60 character problem has 2^{60} subsets of characters. Even if each was resolved in the store, and the lookup time was only 1ns, the total time would still be more than 36 years. Because we want to solve problems with significantly more than 60 characters within a reasonable amount of time, we must find a way to avoid all computation for an enormous number of subsets.

The lattice in Figure 2 provides the inspiration. Notice that, by Lemma 1, if a subset is not compatible, all of its descendents in the lattice must also be incompatible. Similarly, if a subset is compatible, all of its ancestors must be compatible. We can begin at either the top or the bottom of the lattice and continue along each branch until we reach a failure or a success, respectively.

From the lattice, we remove edges to obtain the search tree shown in Figure 10. The tree corresponds to a bottom-up search: we begin with small subsets and progress to larger subsets. The search tree for top-down search is the mirror image of the bottom-up tree. Trees with this structure are known as *binomial trees* [5, 9]. Binomial trees have the recursive structure shown in Figure 11. Figure 12 shows an example with 4 characters.

Notice that a depth-first and right-to-left search of the tree in Figure 12 visits the subsets in lexicographic order. This order visits a subset only after visiting all subsets of that subset. Therefore, the use of the store for failures is perfect in the sense that no subset that has a subset that failed requires the perfect phylogeny procedure. For the same reasons, checking the store for supersets that have succeeded is unnecessary. Similar observations apply to top-down search.

Based on experimental evidence, we have chosen bottom-up search. Intuitively, we believe that most subsets of characters are incompatible, so bottom-up search finds dead ends more

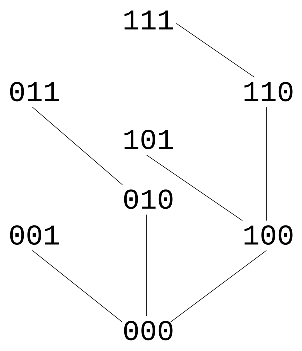

Figure 10: The search tree.

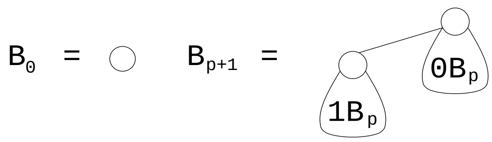

Figure 11: The recursive structure of binomial trees.

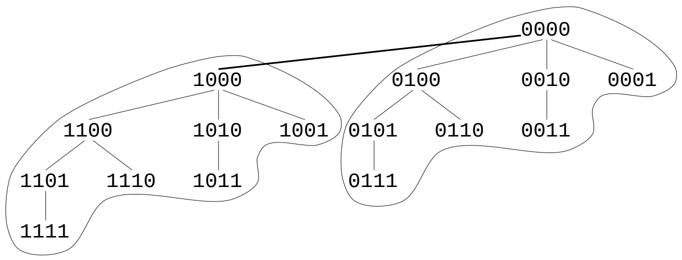

Figure 12: A binomial tree with 4 characters.

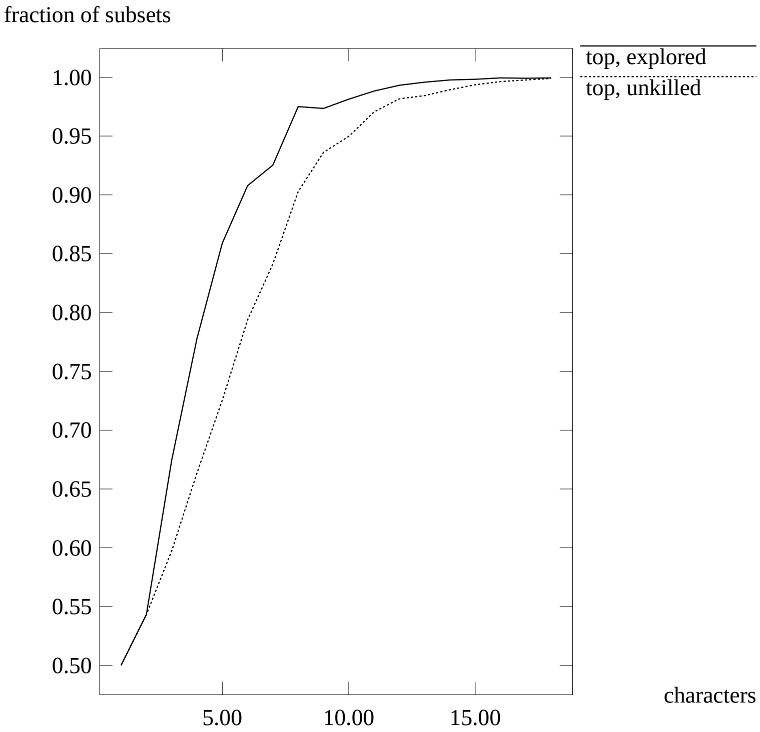

Figure 13: Fraction of subsets explored for top-down search.

quickly and explores fewer nodes of the tree. Experimental data supports our intuition. We compared the top-down and bottom-up approaches for 15 problems with 14 species and 10 characters, all taken from mitochondrial third positions in the D-loop region [4]. The top-down version explored an average of 1004 subsets, and the bottom-up version explored an average of 151.1. Note that the search tree contains only 1024 nodes. Furthermore, only 3.22% of subsets were resolved by accessing the store for top-down search, but for bottom-up, 44.4% were resolved in the store. The difference is more severe for problems with more characters, as shown in Figures 13 and 14. Bottom-up search is the clear winner.

Next, we compare the different strategies for attacking the character compatibility problem: enumerating the subsets without looking in the store (enumnl), enumerating and looking in the store (enum), bottom-up search without looking in the store (searchnl), and bottom-up search with store lookups (search). The timings shown in Figures 15 and 16 were taken on an HP715/64, using the same data sets as above. The data show the superiority of bottom-up search with store lookups.

Also, we observe here that the running time appears to be exponential in the number of characters. We expect this for the simple enumerate without lookup strategy, because we must examine all subsets of the set of characters. Unfortunately, the improved search strategies also have exponential running times. The choice of search strategy can only improve the running time by a constant factor.

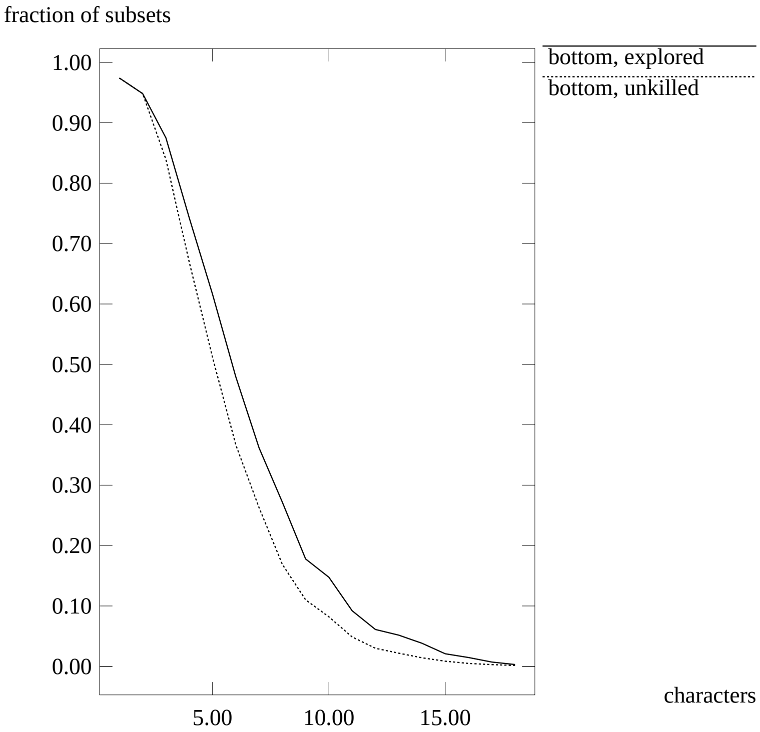

Figure 14: Fraction of subsets explored for bottom-up search.

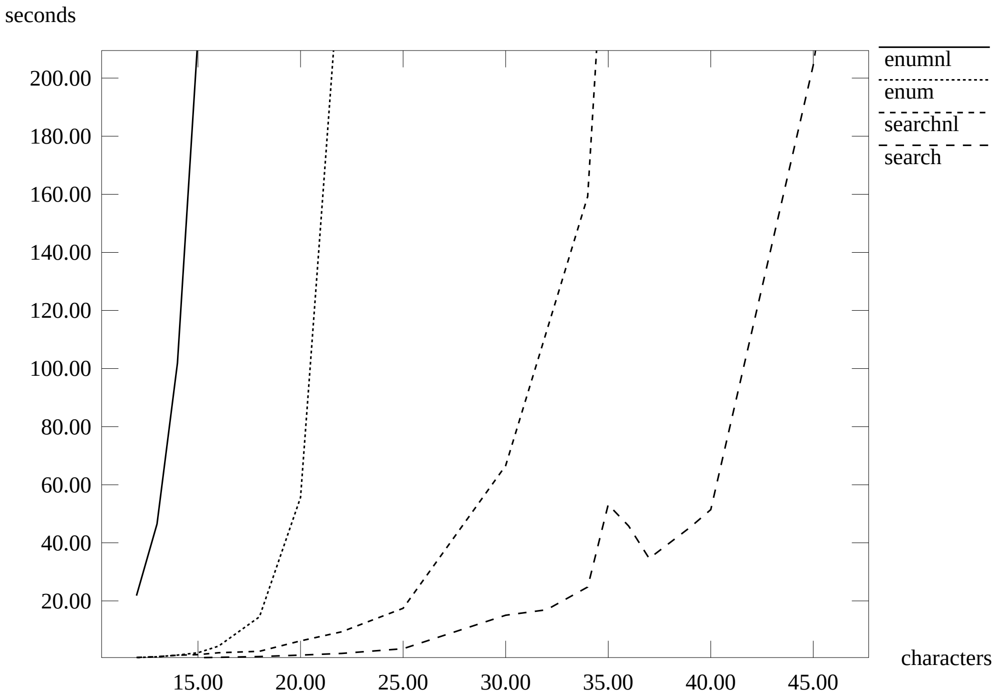

Figure 15: Times for the various search strategies.

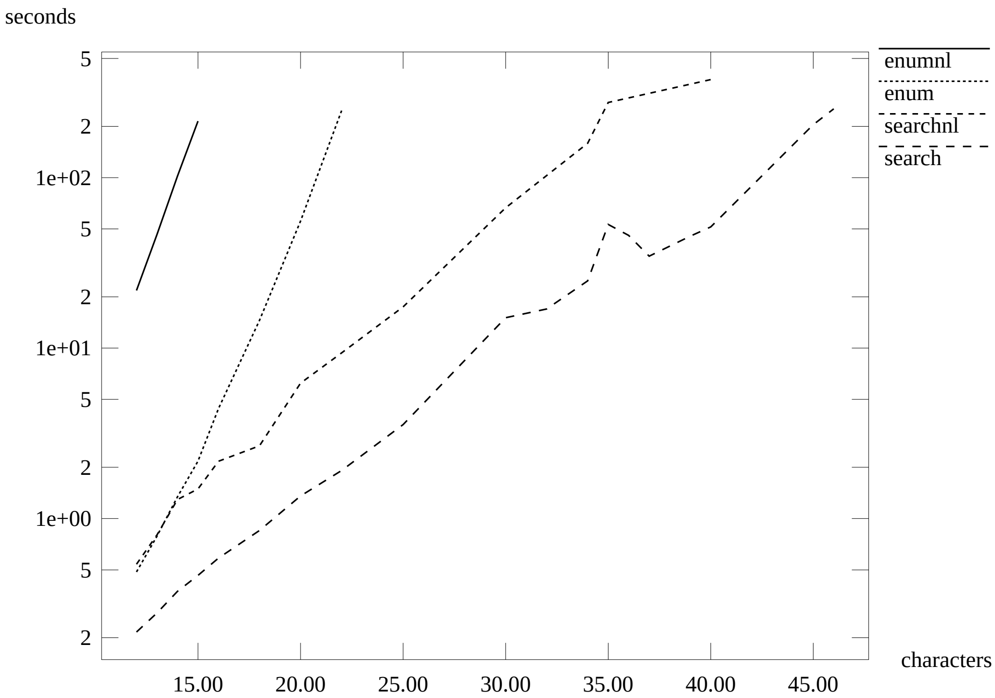

Figure 16: Times for the various search strategies on a logarithmic scale.

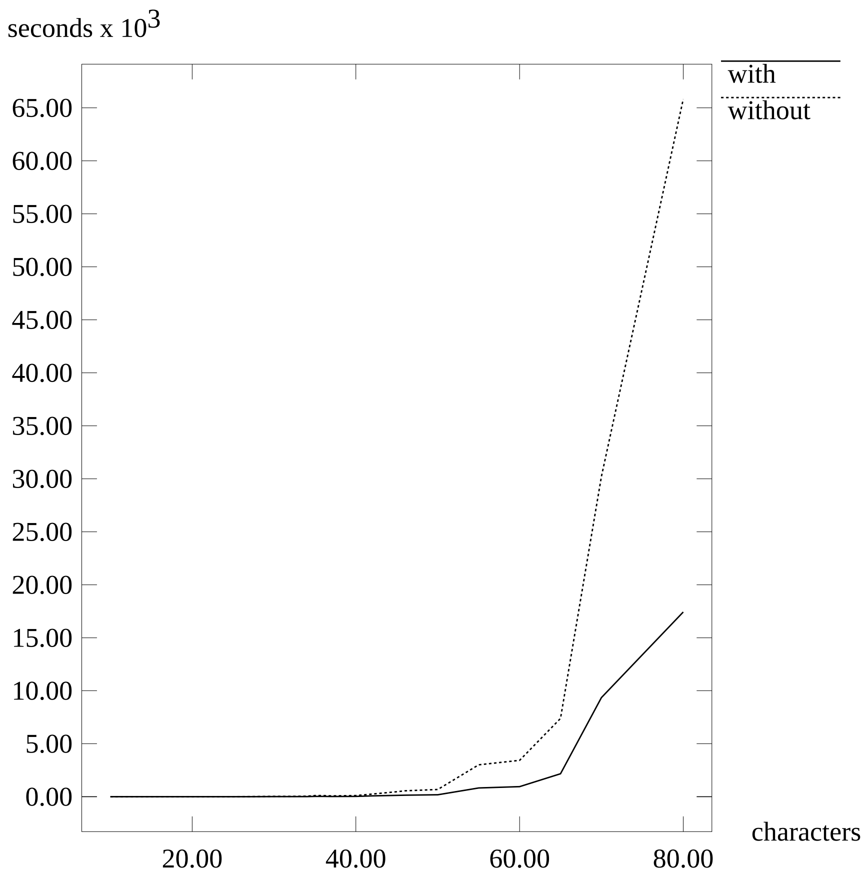

Figure 17: Average times with and without vertex decompositions.

4.2 Vertex Decomposition

As noted in Section 3, vertex decomposition is unnecessary for the correctness of the perfect phylogeny algorithm. Figure 17 shows the performance of the character compatibility implementation with and without vertex decompositions, and Figures 18 and 19 show the number of vertex and edge decompositions, respectively, found by each of the implementations.

4.3 Representation of the Store

Until now, we have considered the store at an abstract level. In this section, we discuss the design decisions in our implementation.

Storing successes and storing failures require different operations, so we separate them logically into two abstract data types, a FailureStore and a SolutionStore. Because we have chosen bottom-up search, we only use the FailureStore, as noted above.

The FailureStore must support the following operations:

- Insert(S'): insert a new set S' of characters into the store
- DetectSubset(S'): determine if any subsets of S' are in the store.

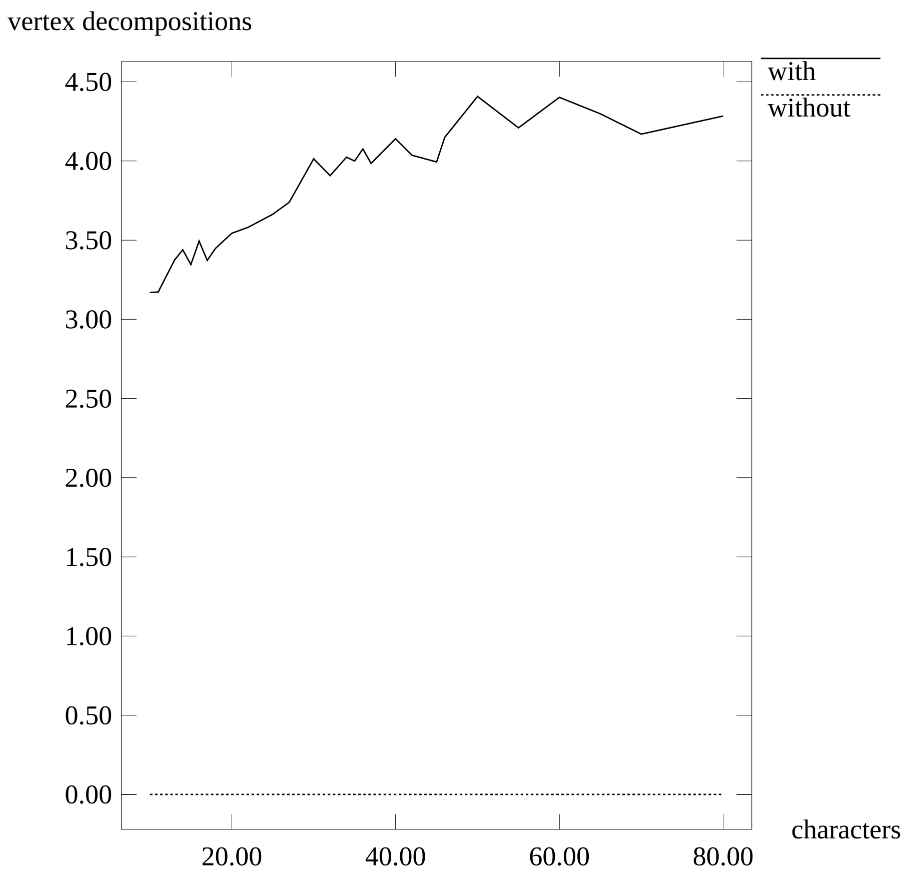

Figure 18: Average number of vertex decompositions found per perfect phylogeny problem.

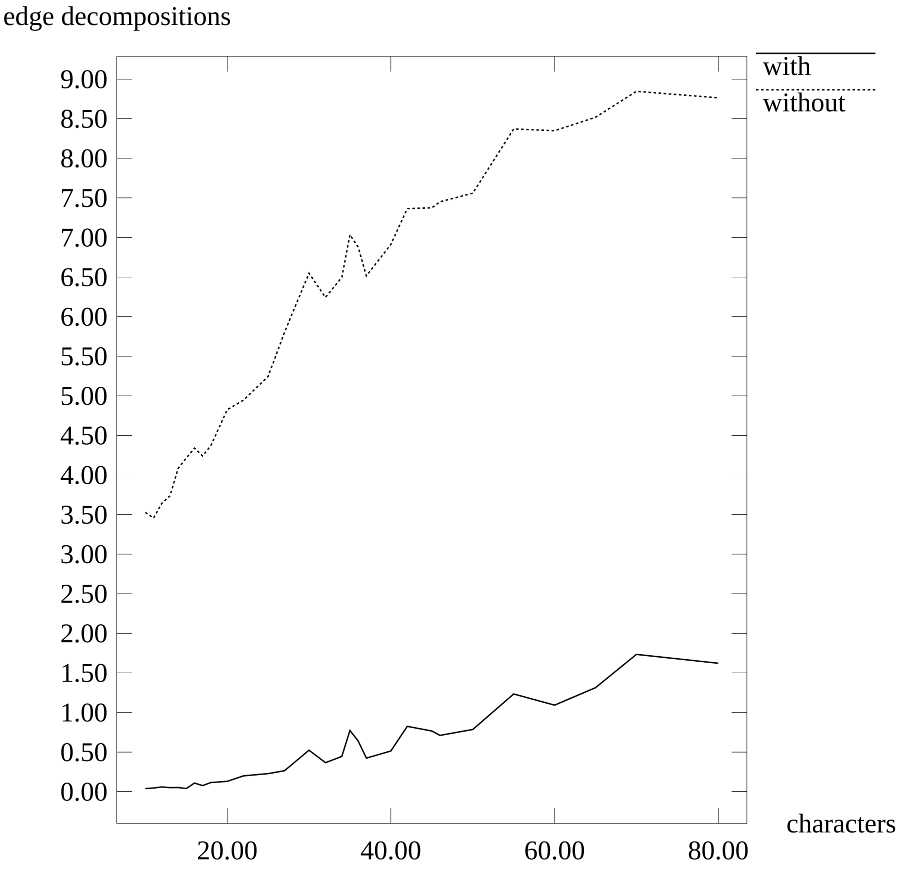

Figure 19: Average number of edge decompositions found per perfect phylogeny problem.

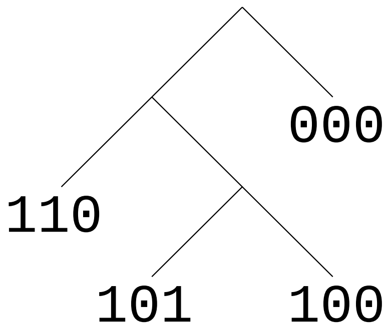

Figure 20: An example trie.

We consider two representations for the FailureStore: a linked list and a trie, each containing sets of characters.

The linked list is a simpler implementation: Insert simply adds the set to the tail of the list, and DetectSubset scans the list looking for subsets. However, because the number of failures can be large, we require that Insert remove any supersets of the inserted set. In other words, we maintain the invariant that no member of the FailureStore is a proper superset of another. The invariant reduces memory requirements as well as the number of sets examined during DetectSubset operations. Note that removing the supersets does not affect the outcome of subsequent DetectSubset operations. The supersets are redundant because the inserted set is a subset of any set of which a removed superset was a subset. Nevertheless, bottom-up, right-to-left search explores the subsets in lexicographic order, so it visits a set only after all its subsets, and the FailureStore never contains any supersets of an inserted set. This search pattern avoids the expense of removing supersets during Insert.

The trie implementation of the FailureStore is similar, but benefits from the fact that the trie structure reflects, to some extent, the relation between subsets. A trie is a tree where a subset is stored in a leaf specified as follows: Representing a subset by a bit vector, where a bit is 1 if the corresponding element is in the subset, and 0 otherwise, we locate a particular subset by starting at the root and, at the nth level, choosing the left subtree if the nth bit is 1 and the right subtree otherwise. Figure 20 shows a trie representing the set of subsets $\{\{\}, \{0\}, \{0, 2\}, \{0, 1\}\}$, which, using bit vector representation, is $\{000, 100, 101, 110\}$.

We can exploit the structure of the trie for detecting subsets with the following observation: If the first bit of a set is 0, then all of its subsets must be in the right subtrie. This applies at each level of the trie, so that, in effect, we only need to search a trie with height equal to the number of elements in the set, instead of the total number of characters. We believe that the trie has a significant advantage because large sets of characters are typically incompatible, so bottom-up search usually calls DetectSubset with small sets. Searching for supersets of a set may be expensive, because we must search a trie with height equal to the number of elements absent from the set, which may be large, although not as large as the number of sets in the FailureStore. But, as discussed above, bottom-up, right-to-left search obviates this operation.

We base our final decision on timing our benchmark suite. Figures 21 and 22 show the results on an HP712/80. The trie has approximately a 30% performance advantage for large problems. In the parallel implementation, however, we expect the margin to be larger because we can no longer guarantee to visit sets in lexicographic order, so removing supersets during Insert is necessary.

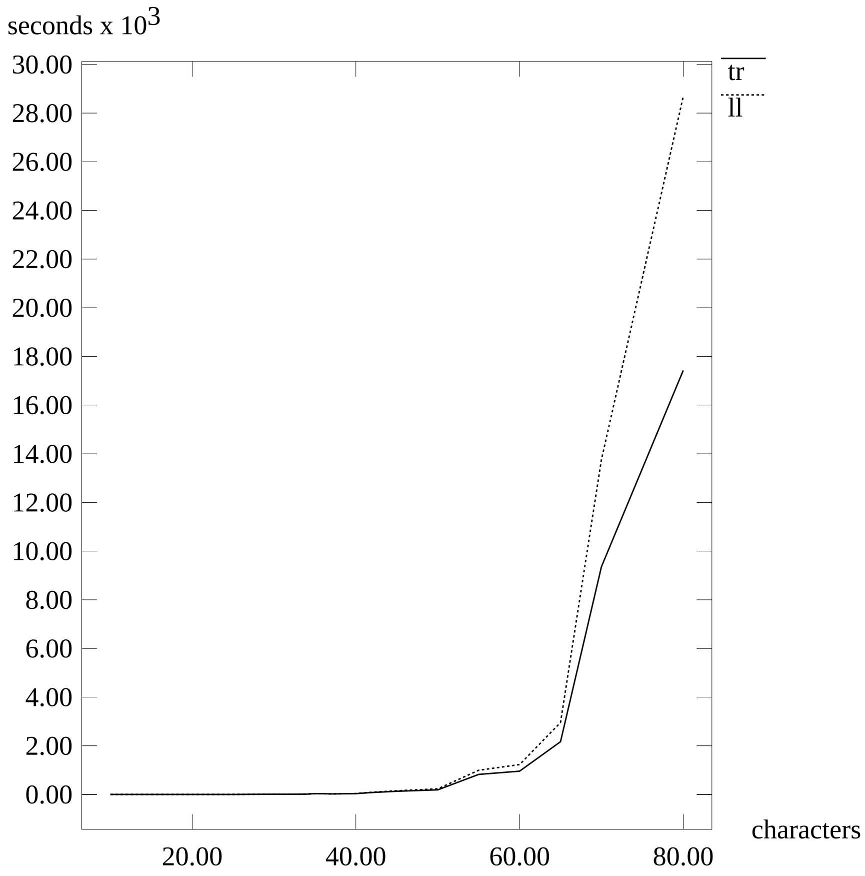

Figure 21: Performance of trie and linked list FailureStores.

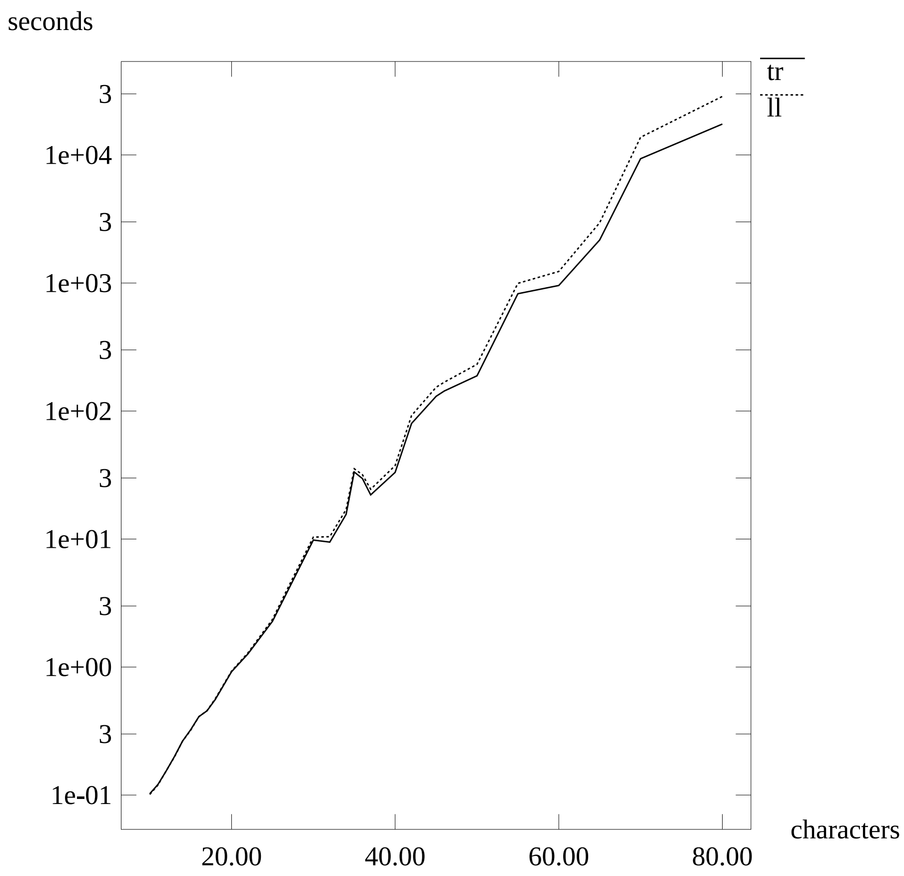

Figure 22: Performance of trie and linked list FailureStores on a logarithmic scale.

5 Parallel Implementation

Finally, we present the design of our parallel implementation.

5.1 Sources of Parallelism

We identify two sources of parallelism in the program. The top level of parallelism comes from the character compatibility problem: solving the perfect phylogeny problem for many different subsets of the character set. These tasks are independent, except for their effect on the FailureStore. The second, lower level of parallelism is within the perfect phylogeny procedure, which uses a divide-and-conquer algorithm. After a vertex decomposition, for example, the procedure recurses on the two subsets, which are two independent tasks.

Our implementation takes advantage of the first source of parallelism only, because we believe that the number of tasks is large enough to sustain a large number of processors. Figure 23 shows the average number of subsets explored for various problem sizes, and Figure 24 shows the average number of subsets that were not resolved in the FailureStore. Even for the moderate-sized problems shown, the number of tasks is enormous, and appears to grow exponentially with the number of characters, as expected. Because we want to solve problems with hundreds or thousands of characters, the parallelism at this level seems to be sufficient.

We associate each task with some computation and some data. The data required for the perfect phylogeny problem is the subset of characters and the character vectors for the original set of species. Because each task uses the character vector for each species, we replicate these data on each processor. Therefore, communicating a task between processors only requires sending the subset of characters. We represent a subset by a bit vector, requiring one bit for every character in the original set and a small amount of header data. Even a 100-character problem needs only five 32-bit words for each task.

Figure 25 shows that the average time per task is on the order of 500μs for an HP712/80. We take advantage of the coarseness of the tasks in designing our implementation.

The first step in producing a parallel implementation from our sequential one is modifying the search procedure to take advantage of the parallelism among tasks. We replace the recursive procedure with an iterative procedure that uses a task queue, and rely on the task queue to distribute the tasks among the processors. We want the task queue to provide dynamic load balancing because the search tree, as well as the amount of computation required for each node, is unknown until runtime. In addition, we prefer a distributed task queue, so that the queue is not a performance bottleneck. The task queue data structure from Multipol [10] meets our requirements, and is our choice for our implementation.

Each processor executes a loop consisting of dequeuing a task from the task queue, executing the task, and enqueuing any new tasks generated. A task corresponds a particular subset of characters, and executing the task consists of determining if the subset is compatible. In the sequential implementation, we used the FailureStore to improve performance. To complete the parallel implementation, we implement the FailureStore as a distributed data structure.

5.2 Distributing the FailureStore

Section 4 showed the importance of the FailureStore to sequential performance. We would like the parallel implementation to reap similar benefits. In this section, we evaluate three approaches to implementing the FailureStore.

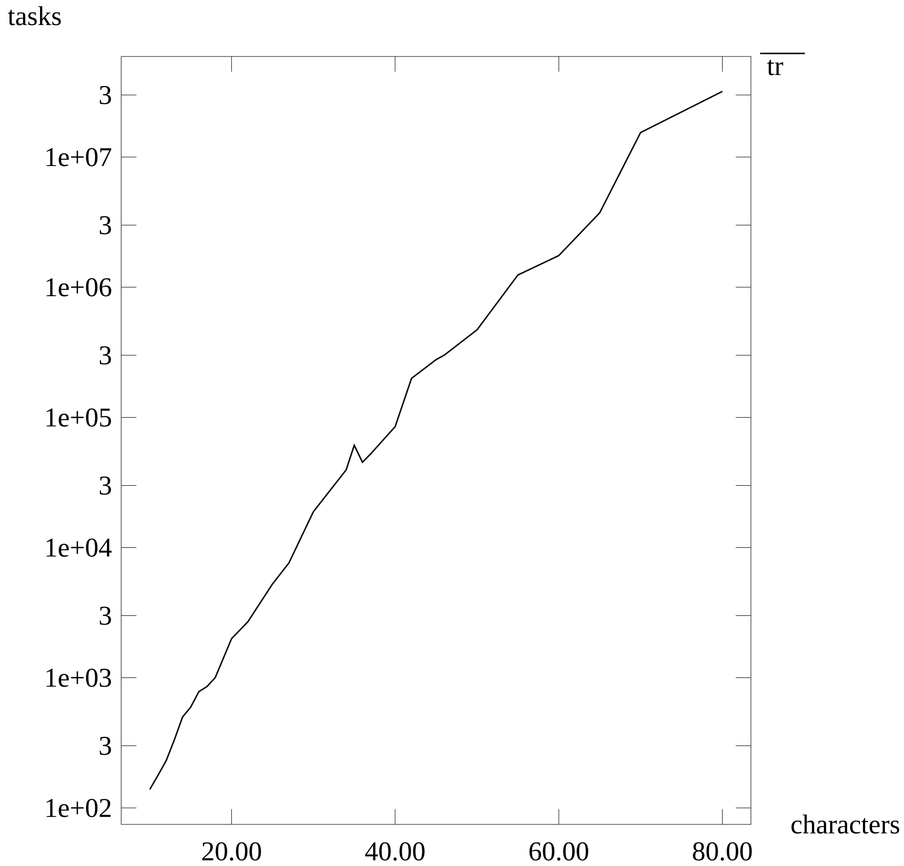

Figure 23: Average number of tasks, logarithmic scale.

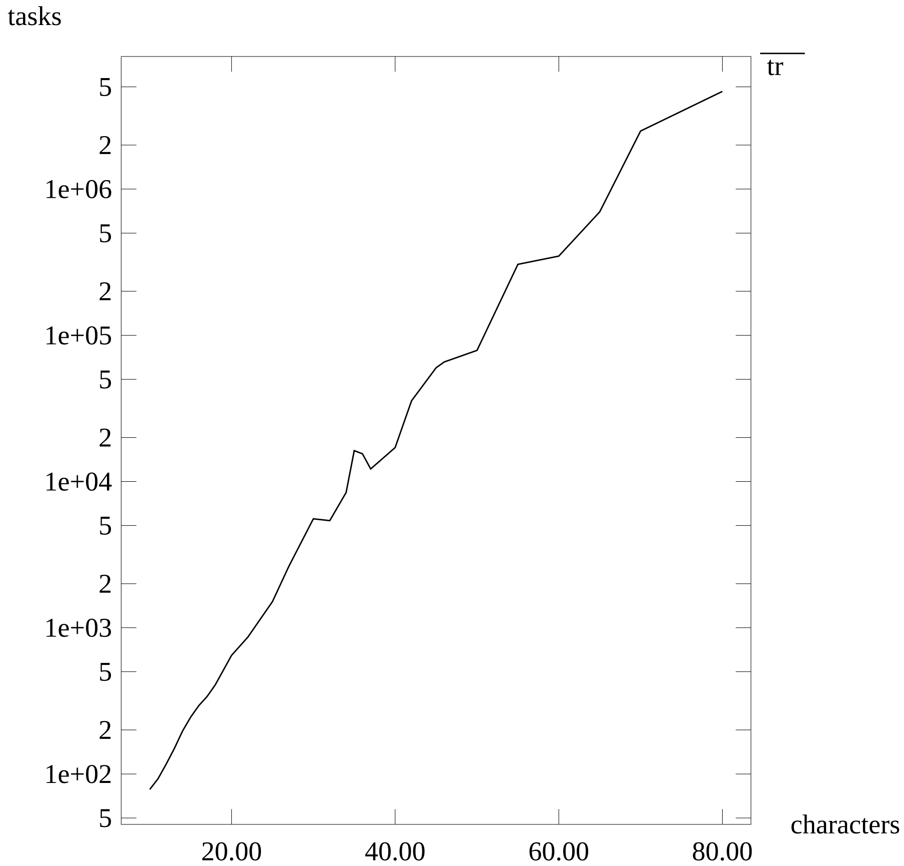

Figure 24: Average number of tasks not resolved in the FailureStore, logarithmic scale.

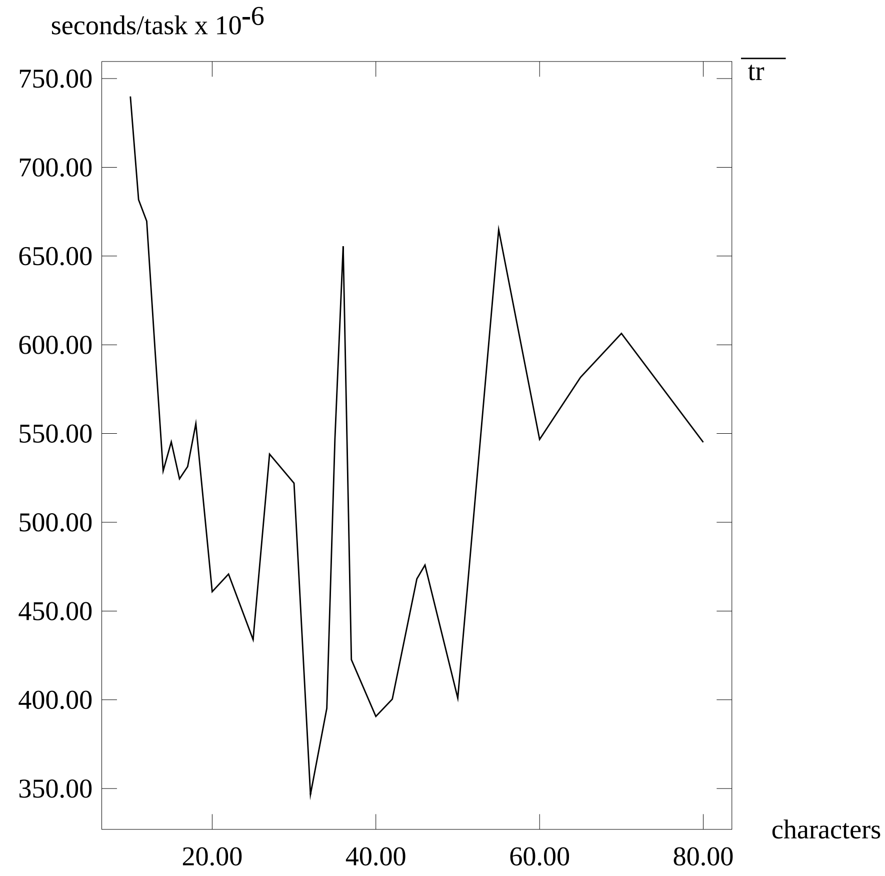

Figure 25: Average time per task.

The simplest approach is to keep a standard trie on each processor. Insert operations insert into this trie, and DetectSubset searches only the local trie. This implementation may lead to redundant work, but it is correct. Processors do not use information about failures discovered by other processors, so a processor may call the perfect phylogeny procedure on a subset which has a subset that another processor has already determined to be a failure. The perfect phylogeny procedure proceeds unaffected, however, and determines that the subset is a failure, so the processor obtains the correct answer and does not explore that branch of the search tree further. The cost of the lack of information is no more than the cost of a call to the perfect phylogeny procedure.

To reduce the amount of redundant work, the processors must communicate information from their FailureStores to other processors. One method is to periodically send a random element from the local trie to another processor. The primary feature of the randomized method is lack of synchronization.

An alternative method is to periodically synchronize and communicate all information in local tries to all processors in a global reduction. This method has a tradeoff between completeness of information on each processor and overhead: Communicating more often requires more time for communication and synchronization, but gives the processors more information so that they can increase the number of subsets resolved in the FailureStore, therefore reducing computation time.

We now present timing results for each of the three implementations. The benchmarks are 40 character sections of the same mitochondrial third positions in the D-loop region [4]. Figure 26 shows the times on a 32-node CM-5 and Figure 27 shows the speedups. Note that for small numbers of processors, the unshared and random techniques achieved superlinear speedup. This is a common effect in search problems, and results from the fact that with more than one processor, some failures may be found more quickly, allowing other processors to avoid computation that is necessary in the sequential case. Nevertheless, the synchronizing combining method seems to perform best as the number of processors increases.

Figure 28 shows the fraction of subsets that were resolved in the FailureStore. The synchronizing method maintains a high rate, but the other two methods suffer from a lack of complete information on each processor, resulting in a higher overall time.

Our goal in this section was to use parallel processing to solve large problems in a reasonable amount of time. Unfortunately, we face two obstacles. First, the efficiency of our implementation is only about 2/3 for 32 processors, and it seems to be decreasing. For larger problems, however, we expect the efficiency to be higher because of the abundance of tasks. We could not plot a speedup curve for larger problems, however, because of memory limitations on the CM5.

The second obstacle is memory limitations. The three implementations of the FailureStore replicate the data on the processors, which restricts the maximum problem size we can solve. Perhaps a truly distributed FailureStore would remedy the problem.

6 Conclusion

We have described the design and implementation of a parallel algorithm for solving the phylogeny problem. The journey has been difficult, each step has been instructive.

The first step was sequential the algorithm. We use the character compatibility method, which searches a large set to find the best phylogeny. We introduced techniques for bounding this search, and showed their substantial impact on performance.

Next, we considered the perfect phylogeny problem, a subproblem of character compatibility.

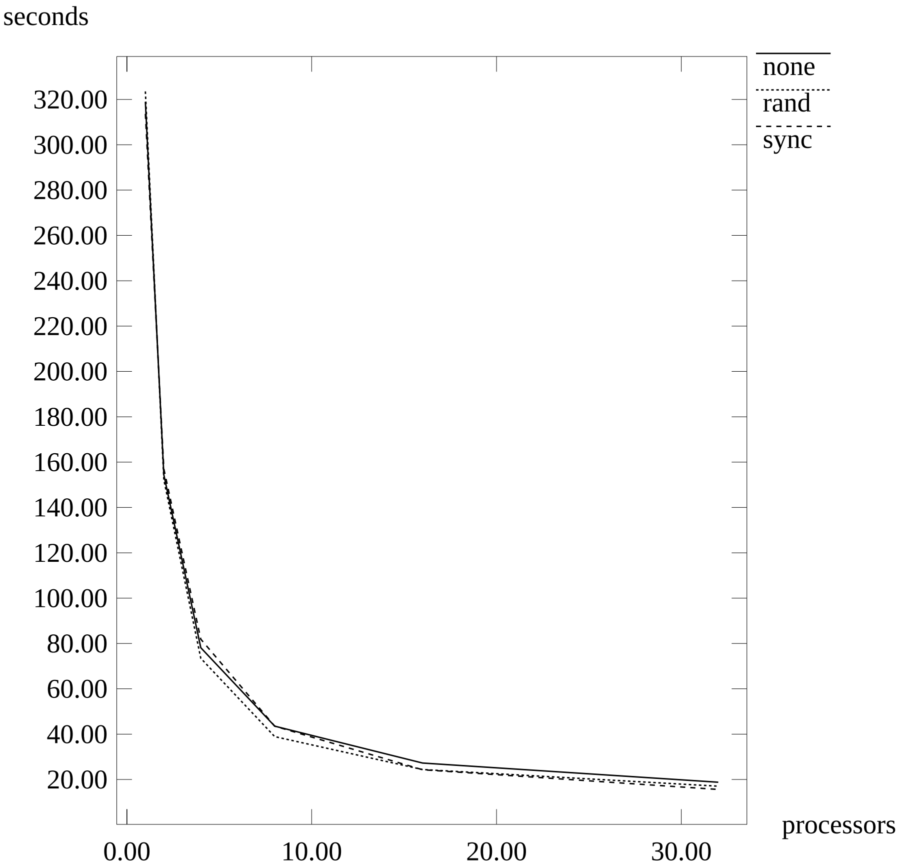

Figure 26: Time vs. processors.

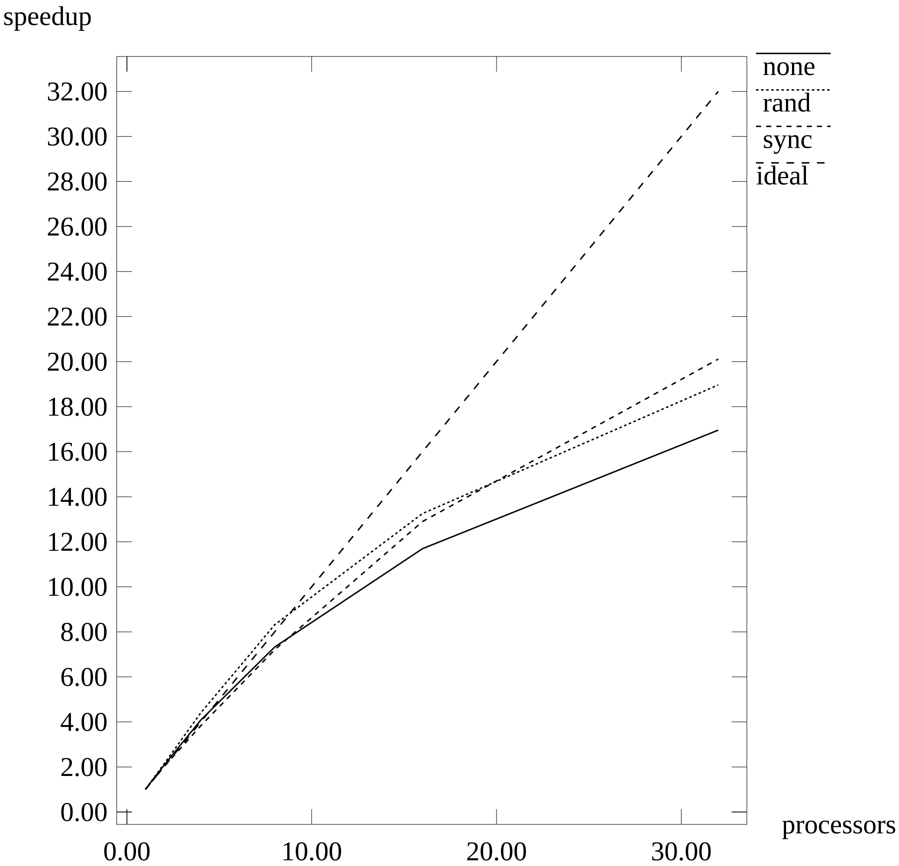

Figure 27: Speedup vs. processors.

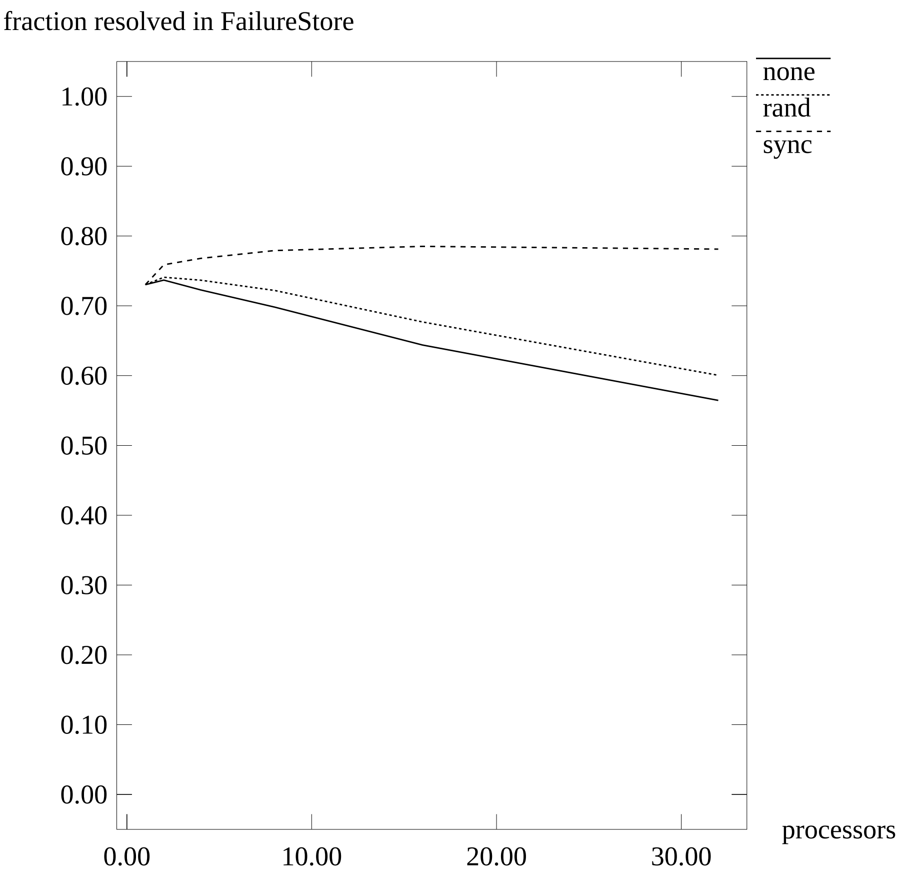

Figure 28: Fraction of subsets resolved in the FailureStore vs. processors.

Our implementation uses the algorithm of of Agarwala and Fernández-Baca [1], implemented according to a suggestion from Lawler [6]. We believe our description and proof of correctness of the algorithm to be simpler than that in the original paper [1]. We also examined several aspects of the performance of our implementation.

Guided by our measurements of the sequential implementation, we developed an initial parallel version based on two data structures. The task queue from Multipol [10] distributes the tasks and maintains load balance, and the FailureStore, represented as a distributed trie, manages the sharing of information among processors. We studied three implementations of the FailureStore and found that the implementation that synchronized periodically to communicate information to all processors was the best.

Our goal throughout was to solve larger problems in a reasonable amount of time. Our parallel implementation achieves reasonable speedups for problems which fit in the memory of a single processor, and we expect better efficiency for larger problems. Section 5 suggested several improvements which we plan to implement in the near future. With the power of parallel computing, solving large phylogeny problems is finally within reach.

References

[1] R. Agarwala and D. Fernández-Baca. A polynomial-time algorithm for the perfect phylogeny problem when the number of character states is fixed. In *Proceedings of the 34th Annual Symposium on the Foundations of Computer Science*, pp. 140-147, 1993.

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

[3] J. Felsenstein. Numerical methods for inferring evolutionary trees. *Q. Rev. Biol*, 57:379-404, 1982.

[4] M. Hasegawa, H. Kishino, K. Hayasaka, and S. Horai. Mitochondrial DNA evolution in primates: transition rate has been extremely low in the lemur. *J. Molecular Evolution*, 31(2):113-21, 1990.

[5] J.M. Hullot. Associative-commutative pattern matching. 5th IJCAI, Tokyo, Japan, 1979.

[6] E.L. Lawler. Personal communication. August 1993.

[7] W.J. Le Quesne. A method of selection of characters in numerical taxonomy. In *Syst. Zool.*, 18:201-205, 1969.

[8] F.R. McMorris, T.J. Warnow, and T. Wimer. Triangulating vertex colored graphs. In *Proceedings of the 4th Annual Symposium on Discrete Algorithms*, Austin, Texas, 1993.

[9] J. Vuillemin. A data structure for manipulating priority queues. *C. ACM*, 21(4), 1978.

[10] K. Yelick, S. Chakrabarti, E. Deprit, J. Jones, A. Krishnamurthy and C. Wen. Data structures for irregular applications. DIMACS Workshop on Parallel Algorithms for Unstructured and Dynamic Problems, 1993.