Draft -- Final version to appear in the proceedings of the *Supercomputing '95*.

Parallelizing the Phylogeny Problem

Jeff A. Jones   Katherine A. Yelick
Computer Science Division
University of California, Berkeley *

Abstract

The problem of determining the evolutionary history of species in the form of phylogenetic trees is known as the phylogeny problem. We present a parallelization of the character compatibility method for solving the phylogeny problem. Abstractly, the algorithm searches through all subsets of characters, which may be traits like opposable thumbs or DNA sequence values, looking for a maximal consistent subset. The notion of consistency in this case is the existence of a particular kind of phylogenetic tree called a perfect phylogeny tree.

The two challenges to achieving an efficient implementation are load balancing and efficient sharing of information to enable pruning. In both cases, there is a trade-off between communication overhead and the quality of the solution. For load balancing we use a distributed task queue, which has imperfect load information but avoids centralization bottlenecks. To prune the search space, we use the following property: If a perfect phylogeny tree does not exist for some set of characters, then none exists for any superset of that set. This is implemented by searching the power set starting with the smallest sets, and storing failures in an efficient distributed trie. The resulting program shows speedups of 50 on a 64-processor CM-5.

1 Introduction

The problem of determining the evolutionary history for a set of species, known as the *phylogeny problem*, is fundamental to molecular biology. Evolutionary history is typically represented by a *phylogenetic tree*, a tree of species 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.

Methods for solving the phylogeny problem include parsimony, compatibility, maximum likelihood, and distance matrix methods [5]. Unfortunately, with any of these methods, deriving phylogenetic trees is often prohibitively expensive. In this paper we describe the parallelization of a solution to the phylogeny problem; it is the first parallel implementation of this algorithm and is based on our own sequential implementation which is significantly faster than previous sequential implementations.

*This work was supported in part by the Advanced Research Projects Agency of the Department of Defense monitored by the Office of Naval Research under contract DABT63-92-C-0026, by the Department of Energy grant DE-FG03-94ER25206, by the National Science Foundation as a Research Initiation Award (number CCR-9210260), and as an Infrastructure Grant (number CDA-8722788), The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Our algorithm is based on character compatibility [10]. The problem statement is defined formally in Section 2 and a sequential algorithm outlined in Section 3. Character compatibility is essentially a search problem and as such has two of the classic problems of parallel search. First, the search tree unfolds dynamically with unpredictable structure and cost; each node in the search tree is a parallel task to be assigned to processors while optimizing for both locality and load balance. Second, as with many search problems, pruning of the search space is essential to performance, and pruning in a parallel implementation requires sharing information about the results of the search so far. In something like a game tree search with alpha-beta pruning, the shared information may be small. In our case, the shared state is a representation of the result (success or failure) of every node searched so far. Fortunately, there is sufficient structure in the state that its representation is not prohibitively large.

Because of pruning and irregular task times, the performance of character compatibility is difficult to predict. Nevertheless, we show how measurements of the sequential algorithm with typical input values are used to motivate the parallel design. In Section 4 we describe some of the important performance characteristics of the algorithm and survey its sources of parallelism.

The parallelization is done by replacing two of the main data structures from the sequential algorithm, a queue and a trie, with distributed versions. The queue holds search tree tasks and is replaced by a distributed task queue for dynamic load balance, described in Section 5. The trie holds the known successes and failures and is represented by a lazily replicated trie, described in Section 6. Four different implementations of the trie were considered, and we provide performance numbers that demonstrate the sensitivity to design choices, and show that for the best choice, the overall speedups are quite good. Section 7 presents our conclusions.

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*, or perfect phylogeny for short, for a set of species and a given set of characters for those species is a phylogenetic tree compatible with all the characters. Formally, we define a perfect phylogeny 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, $0 \leq i \leq k$, $u_i[c] = u_0[c]$*

species	characters
u	1 1 4
v	1 2 1
w	2 1 2

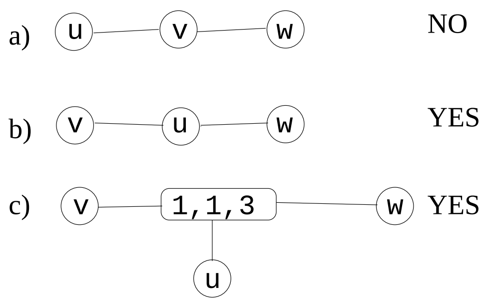


Figure 1: Examples of perfect phylogenetic trees.

species	characters
u	1 1
v	1 2
w	2 1
x	2 2

Figure 2: Set that does not have a perfect phylogeny.

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*. This is equivalent to stating that a set of characters is *compatible* if there exists a perfect phylogeny for the set of species that is compatible with every member of the set.

To illustrate the definition, consider Figure 1, which shows 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 $v[2] \neq u[2]$. 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.

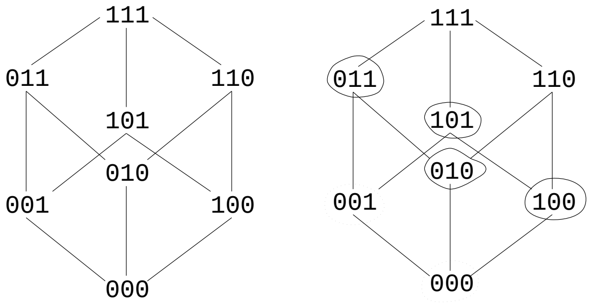
In contrast, trees b and c are perfect phylogenies because they satisfy the three conditions of the definition. Notice that the perfect phylogeny for a given set of species may not be unique. Also, tree c contains the species $[1, 1, 3]$, which was not a member of the original set. The tree is still a perfect phylogeny, however, because each leaf appears 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.

Ideally, we would like to find a perfect phylogeny for our original set of species with the entire set of characters. Unfortunately, this does not exist for many sets of species. Figure 2 shows an example. Even adding new internal vertices does not produce a perfect phylogeny. For this reason, the character compatibility problem is to find the maximal compatible subsets. If one of the maximal subsets is sufficiently large, the corresponding perfect phylogeny will be a good estimate of the evolutionary history of the species.

To determine compatibility, we use the algorithm of Agarwala and Fernández-Baca [1], modified according to a suggestion of Lawler [9], as described in our previous work [8]. 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.

species	characters		
u	1	1	1
v	1	2	1
w	2	1	1
x	2	2	1

a) Input species



b) lattice of subsets 3) frontier of solutions

Figure 3: Search space of characters subsets.

Recall that a set of characters is compatible if a perfect phylogeny exists for the given set of species with that set of characters.

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, C_1 must also be compatible with T.
□

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 sets as being partially ordered by the subset relation. For example, Figure 3a shows a set of species with three characters and Figure 3b shows the generic lattice for the three character case. Figure 3c shows the maximal compatible subsets, marked as solid circles on the lattice, and the non-maximal but compatible subsets, marked as dashed circles. In Section 3, we discuss the performance ramifications of this structure.

3 Exploring the Space of Subsets

Recall that the character compatibility problem is the problem of finding the maximal compatible subsets of characters. 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 using Lemma 1.

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 perfect phylogeny procedure each time. This algorithm parallelizes very well: each processor can explore some portion of the subsets independently. Unfortunately, this method is horribly inefficient, 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 a compatible subset of characters as a success and an incompatible subset as a failure. If we find either a superset that is a success

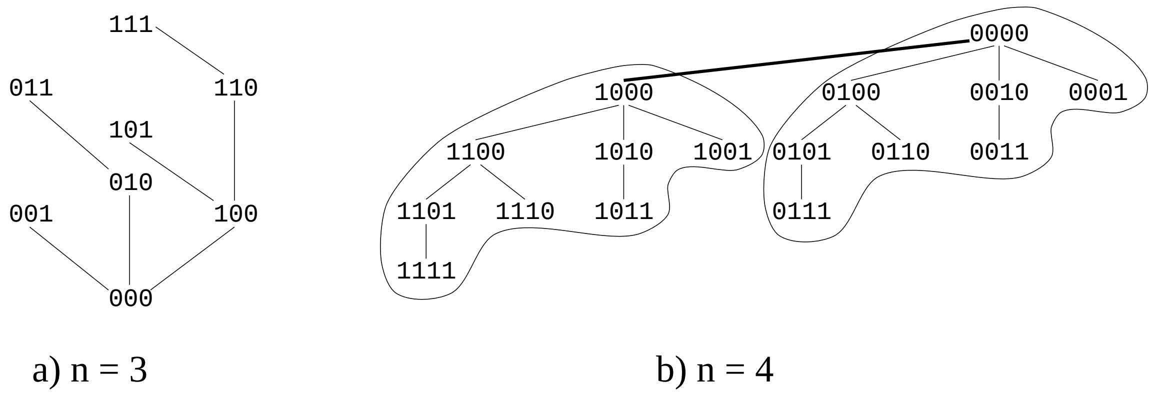


Figure 4: Binomial trees.

or a subset that is a failure, we avoid the cost of the perfect phylogeny procedure. This is a significant improvement, but it comes at the cost of reducing parallelism, because the processors must share the two stores, so they no longer work independently.

This method is insufficient, however, to solve large problems. We must eliminate 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 were 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 3 provides the intuition for pruning the search space. Notice that, by Lemma 1, if a subset is incompatible, all its descendants 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 4a. Trees with this structure are known as *binomial trees* [7, 12]. Figure 4b shows an example with 4 characters. The tree corresponds to a search in which we begin with small subsets and progress to larger subsets. Alternatively, we could begin with large subsets and progress to smaller subsets. This approach, however, is not as efficient [8] because most large subsets will fail, which gives no information about the rest of the subsets. In contrast, many small subsets will fail, allowing us to use Lemma 1 to prune large branches of the search tree.

Notice that we can combine the two techniques above. As we explore the search tree, we refer to the store before calling the perfect phylogeny procedure at each node. In fact, if we proceed depth first and right to left in our search tree, keeping successes in the store is unnecessary because we will visit no set of characters until we have visited all its subsets. This search pattern is very efficient, because the failure store gives perfect information: We will call the perfect phylogeny procedure on no subset of characters which has an incompatible subset.

Not even this combined technique, however, is fast enough for large problems. Therefore, we use parallelism to reduce the running time further. Parallelizing this search technique presents two challenges. First, we must overcome the limitations on parallelism imposed by the structure of the search tree. The search tree reduces the available parallelism by imposing an ordering constraint between parent-child pairs in the tree. Furthermore, the search tree causes a load balancing problem, because neither the shape of the tree that must be explored nor the amount of computation required for each node is known until run time. We consider these problems in Sections 4 and 5.

The second challenge is minimizing the overhead of sharing the failure store and the solution store among processors. There is a trade-off between the overhead of sharing and the cost of visiting nodes that should have been pruned. Section 6 addresses this point.

4 Characteristics of the Application

Two important factors in determining whether a program can be effectively parallelized is the amount of parallelism and the granularity of the parallel tasks. Because tasks will be created and scheduled dynamically, perfect scheduling is not possible; it is therefore important to have many more tasks than processors to avoid idle time. Because most modern parallel machines have powerful nodes and significant communication overhead, the tasks must be fairly coarse-grained.

4.1 Available 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 failure store. The second, lower level of parallelism is within the perfect phylogeny procedure, which uses a divide-and-conquer algorithm.

Our implementation takes advantage of the first source of parallelism only, because the number of tasks appears to be large enough to sustain a large number of processors. Figure 5 shows the average number of subsets explored for various problem sizes, as well as the average number of subsets that were not resolved in the failure store. Even for the moderate-sized problems shown, the number of tasks is enormous, and grows 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 is sufficient.

4.2 Task Granularity

A task in our application is the solution of the perfect phylogeny problem for a given subset of characters. The data required for each task is the subset of characters and the character vectors for the entire set of species. Because each task uses the character vector for each species, we replicate this data on all processors. 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 per task.

Figure 6 shows the distribution of task times for a 60-character problem running on an Hewlett-Packard 712/80. Approximately 80% of the tasks take less than 50 μs. These are not shown in the figure. The time taken by the remaining tasks is on the order of 1 ms. The average task time is about 400 μs, or 32000 cycles. The tasks are very coarse: tens of thousands of cycles requiring only a few words of data. We take advantage of the coarse grain size of the tasks in designing our implementation.

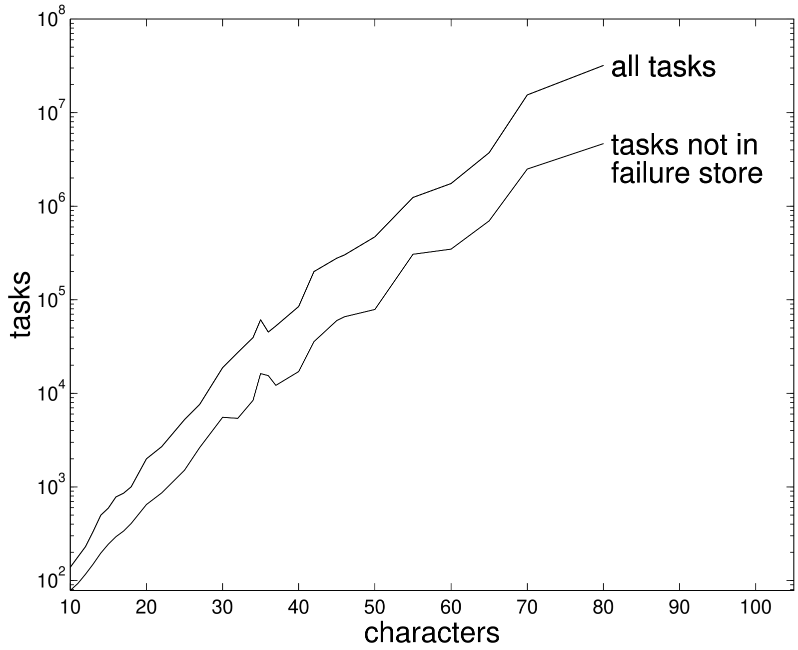


Figure 5: Average number of tasks, logarithmic scale.

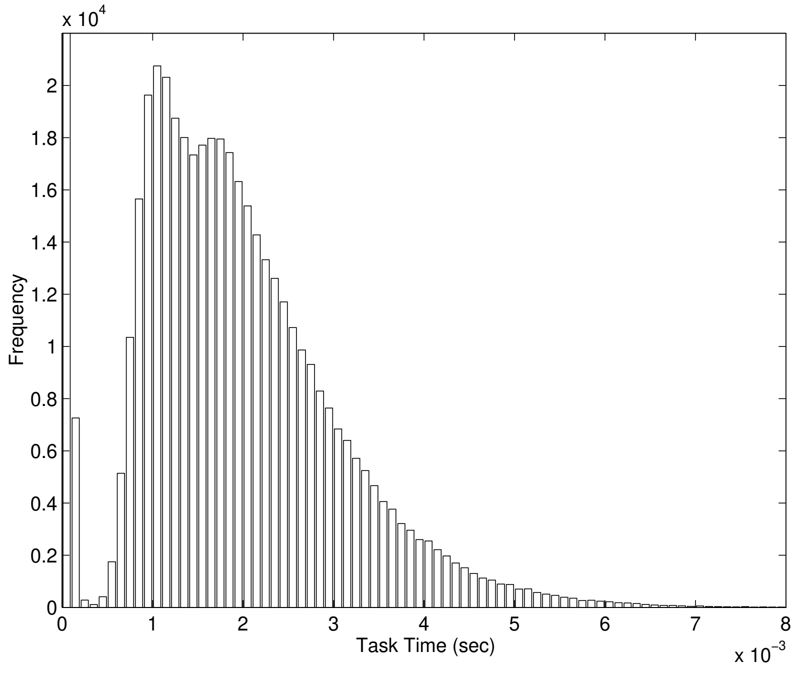


Figure 6: Task time distribution.

5 Distributing the Tasks

To explore the search tree, we use a data structure known as a task queue. The task queue will distribute tasks across the processors to ensure good load balance. This structure supports the following operations:

- Enqueue(T): insert a task T into the task queue.
- Dequeue(T): remove the highest priority task in the the task queue and return it in T.

As mentioned above, each task corresponds to a subset of characters, or, equivalently, a node of the search tree. Each processor executes a loop consisting of dequeuing a task from the task queue, executing the task, and, if the node succeeded, enqueuing the children of the task in the search tree. As mentioned in Section 3, use of the failure store improves performance significantly. To complete the parallel implementation, we implement the failure store as a separate distributed data structure, described in Section 6.

We place three performance requirements on the task queue. First, it must provide good load balancing, even for tasks with unpredictable times and with dependencies between tasks. Second, the overhead of accessing the task queue must be low, and performance should scale with the number of processors. This provides an interesting trade-off, because perfect load information requires centralization, whereas scalability precludes it. The third requirement on the task queue is that it should respect locality. In the phylogeny problem, this means leaving tasks on the processor that created them whenever possible. Although the information necessary to execute the perfect phylogeny procedure is present on all nodes, the information from the distributed failure store may not be. Because the subsets in a subtree of the search tree are all supersets of the root of the subtree, they are likely to benefit from the same elements of the failure store. Therefore, we will be able to resolve more tasks in the failure store if the tasks in the subtree execute on the same processor. Figure 7 shows the difference in running times between an implementation that uses a task queue that respects locality and one that uses a randomized task queue, for three 60-character problems on a 64-processor CM-5.

Along with these performance requirements, we will relax the semantics of the Dequeue operation on the priority queue. To solve a particular phylogeny problem, we must execute some subset of the tasks. Besides the parent-child relationship, the order of execution is not important for correctness. Therefore, the Dequeue operation need not respect the priority of the tasks. This observation allows a simpler, more efficient distributed task queue, because we need not maintain global priority information.

Although priority is not a strict functional requirement, it is important to execute tasks associated with small subsets before larger ones to increase the use of the failure store. As mentioned in Section 3, exploring the search tree depth first and right to left is most efficient, because it maximizes the benefit of the failure store. We use priority to encourage this order.

Input	Task Queue with Locality	Randomized
60:000	89.3	694.1
60:001	180.1	854.9
60:002	221.0	1651.0

Figure 7: Running times (in seconds) for different task queue implementations.

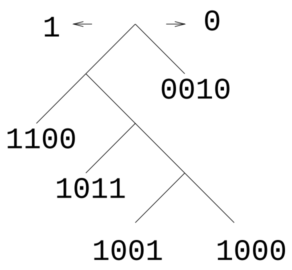


Figure 8: An example trie.

The task queue data structure from the Multipol [13] meets our requirements, and is our choice for our implementation. The task queue has been used in several other applications, including an eigenvalue problem, a symbolic application, and several smaller problems [4].

6 Representation of the Store

In this section, we discuss the design decisions in our implementation of the store abstraction. The same data type can be used to represent both the successes and failures, because they are duals of one another. To avoid confusion, we will discuss only the failure set store in this section. The design of the failure store is more important to performance, since it is used for pruning the search space [8].

The failure store 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.

For parallel performance, we will weaken the semantics of DetectSubset as we did with the Dequeue operation on the task queue. We first describe an efficient sequential data structure and then present four different representations for distributed memory multiprocessors.

6.1 A Sequential Implementation

In previous work, we have determined that a trie is a good data structure for a sequential implementation of the phylogeny problem [8]. A trie is a tree in which each leaf holds one value, in our case a character subset. 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 8 shows a trie representing the set of subsets $\{\{2\}, \{0\}, \{0,3\}, \{0,2,3\}, \{0,1\}\}$, which, using bit vector representation, is $\{0010, 1000, 1001, 1011, 1100\}$. The trie structure reflects, to some extent, the relation between subsets, so we can implement Insert and DetectSubsets efficiently.

To reduce memory requirements, we maintain the invariant that no member of the trie is a subset of another. In fact, in order to correctly execute DetectSubset, we need only keep the minimal subsets of characters, because if a subset is a superset of any member of the failure store, then it is a superset of a minimal element. The benefits of reducing memory requirements include improved cache and virtual memory performance, as well as the ability to complete the program on machines with a small amount of memory and no virtual memory, such as a node of the CM-5.

We present performance results for the parallel implementations of the next three sections below. The benchmarks are two 60 character sections of the mitochondrial third positions in the D-loop region for various species [6]. Figures 9 and 11 show the speedups on a 64-node CM-5 and Figures 10 and 12 show the fraction of subsets that were resolved in the failure store. We will discuss these results in the sections that follow. We were unable to take performance numbers for the fourth, asynchronous version, but plan to do so for the final version of the paper.

6.2 Unshared

The simplest approach to distributing the trie is to keep a standard trie on each processor. Insert operations insert into this trie, and DetectSubset searches only the local trie. Note that DetectSubset is no longer guaranteed to return the correct result. Because no sets on other processors are checked, DetectSubset(S) may return FALSE when a subset of S is stored on another processor. Therefore, this implementation may lead to redundant work, but it is correct. A processor may call the perfect phylogeny procedure on a subset of characters which has a subset that another processor has already determined to be a failure. The perfect phylogeny procedure will proceed unaffected, however, and determine that the subset is a failure, so the processor will obtain the correct answer and will 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.

For both test inputs, the fraction of subsets resolved in the failure store decreases as the number of processors increases, as expected. This method is competitive for small numbers of processors because of the lack of sharing overhead, but the poor performance of the failure store on large numbers of processor dramatically reduces performance, especially for input 60:001.

6.3 Random

To reduce the amount of redundant work, the processors must communicate information from their local tries to other processors. One method is to periodically send a random element from the local trie to another processor. The implementation is relatively simple, and makes use of an explicit message queue on each processor. The advantage of pushing elements, rather than pulling them as would happen in a caching system, is that the sending processor has information about which values need to be propagated. The primary benefit of the randomized method is lack of synchronization. Unfortunately, the amount of information communicated in each message is small, so depending on the frequency of sharing, the program suffers insufficient sharing or excessive communication overhead.

The speedup figures this method is competitive for to input 60:000, but performs extremely poorly for input 60:001. The performance of the failure store explains this difference. For input 60:000, the fraction of subsets resolved in the failure store remains high except for the largest number of processors. For input 60:001, however, random sharing does not maintain sufficient sharing, causing overall performance to plummet.

6.4 Synchronous

An alternative method is to periodically synchronize and communicate all information in local tries to all processors in a global reduction. This reduction is performed in $\log P$ steps, where P is the number of processors. At each step, each processor reads the members of the trie on another processor and inserts them into its trie. The sequence of processors that processor i communicates with is $i \operatorname{xor} 1$, then processor $i \operatorname{xor} 2$, then $i \operatorname{xor} 4$, and so forth up to processor $i \operatorname{xor} P/2$. (Here xor denotes bitwise exclusive or.) All processors synchronize between steps by executing a barrier. The communication pattern is the familiar butterfly reduction tree. After this reduction, all processors have the same information in their local tries.

This method has a tradeoff between completeness of information on each processor and overhead: Communicating more frequently requires more time for communication and synchronization, but gives the processors better information so that they can increase the number of subsets resolved in the failure store, thereby reducing computation time.

The speedup figures demonstrate that this method is superior to the previous two for large numbers of processors because it maintains high performance of the failure store.

An additional benefit of this method is reduced memory requirement. According to intuition, increasing the amount of information stored on each processor should increase the amount of memory required. Because each local trie maintains the invariant that no member is a subset of another, however, the number of elements in the local trie tends to decrease after the combine operation. The random method above does not benefit from this effect, however, because the sharing is very haphazard. This consideration is important for solving large problems on machines with limited memory. In fact, the random method could not run on less than 32 processors because of its memory requirements.

6.5 Asynchronous

The synchronous version makes use of fast sychronization and global communication that was available on the CM-5. On machines like a network of workstations, which do not have this support, we considered a method that performs the global reduction lazily. Instead of synchronizing all processors to perform the reduction, a processor reads the tries from each of the processors that it would have communicated with in the reduction and adds their elements to its own trie. Eventually, this procedure will broadcast all elements to all processors, just as in the synchronous case, but no barrier is necessary.

This method does involve some additional overhead, both in memory and in processing time. Because the processors are not synchronized, a processor will need to preserve old versions of its local trie that other processors are in the process of reading. Also, some processing is necessary to garbage collect these old versions. If the network has reasonably high bandwidth, however, we expect the number of old versions maintained to be small, because the time required to read a trie from another processor will be small.

Another source of overhead is the unsynchronized communication. As noted by Brewer and Kuszmaul [3], on an architecture such as the CM-5, unsynchronized communication can lead to the formation of hot spots. For example, several processors may read from the same processor at the same time, which will cause each of the reading processors to be delayed. The barriers of the synchronous method prevent this occurrence. For this reason, the asynchronous method is well suited to networks on which a barrier is expensive, but may not perform as well as the synchronous method on an architecture such as the CM-5, which has very fast barriers.

As mentioned above, we were unable to take performance numbers for this method. Some

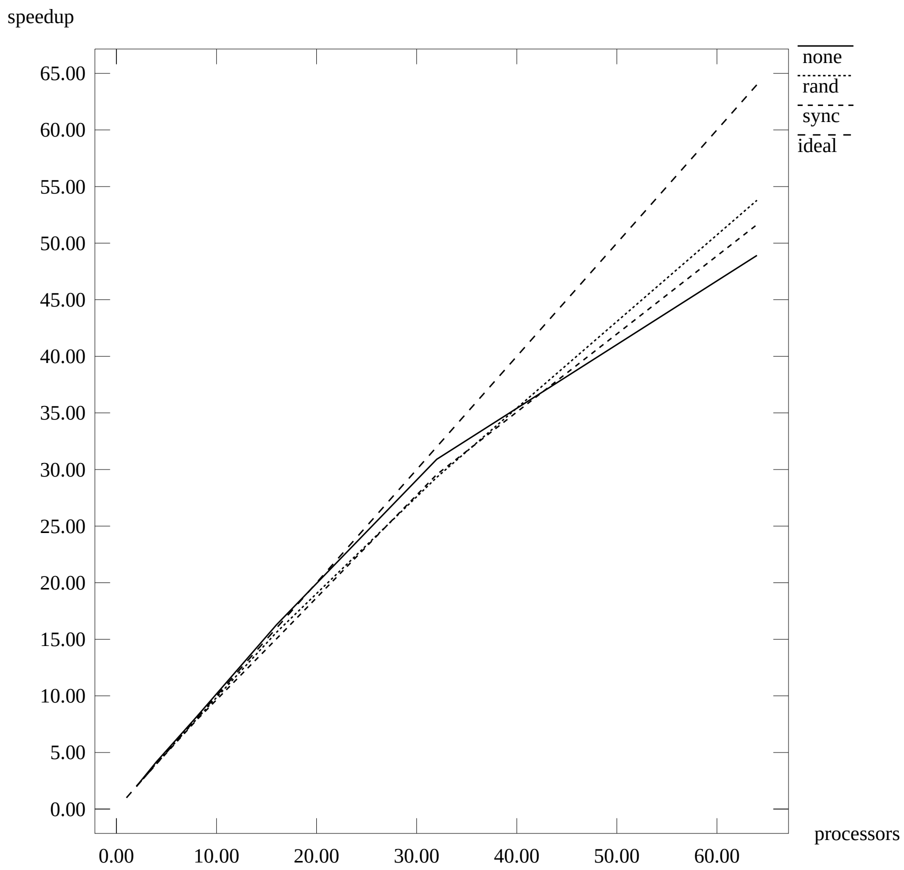


Figure 9: Speedup vs. processors for Input 60:000.

preliminary measurements, however, show performance that is inferior to that of the synchronous version on the CM-5.

7 Conclusion

We have described the design and implementation of a parallel algorithm for solving the phylogeny problem based on the character compatibility method. The application has irregular data structures, asynchronous communication inside the task queue and within some versions of the trie, and an irregular task graph with unpredictable task times due. Guided by our measurements of the sequential implementation [8], we developed a parallel version based on two data structures. The task queue from Multipol [13] distributes the tasks and maintains load balance, and the failure store, represented as a distributed trie, manages the sharing of information among processors. We studied four implementations of the failure store and found that the implementation that synchronized periodically to communicate information to all processors was the best. In addition to the usual amortization of overhead that results from sending a few large messages rather than many small ones, in this application a larger set of values may actually require less space, because the combined the representation may shrink.

Our goal throughout was to solve larger problems in a reasonable amount of time and to

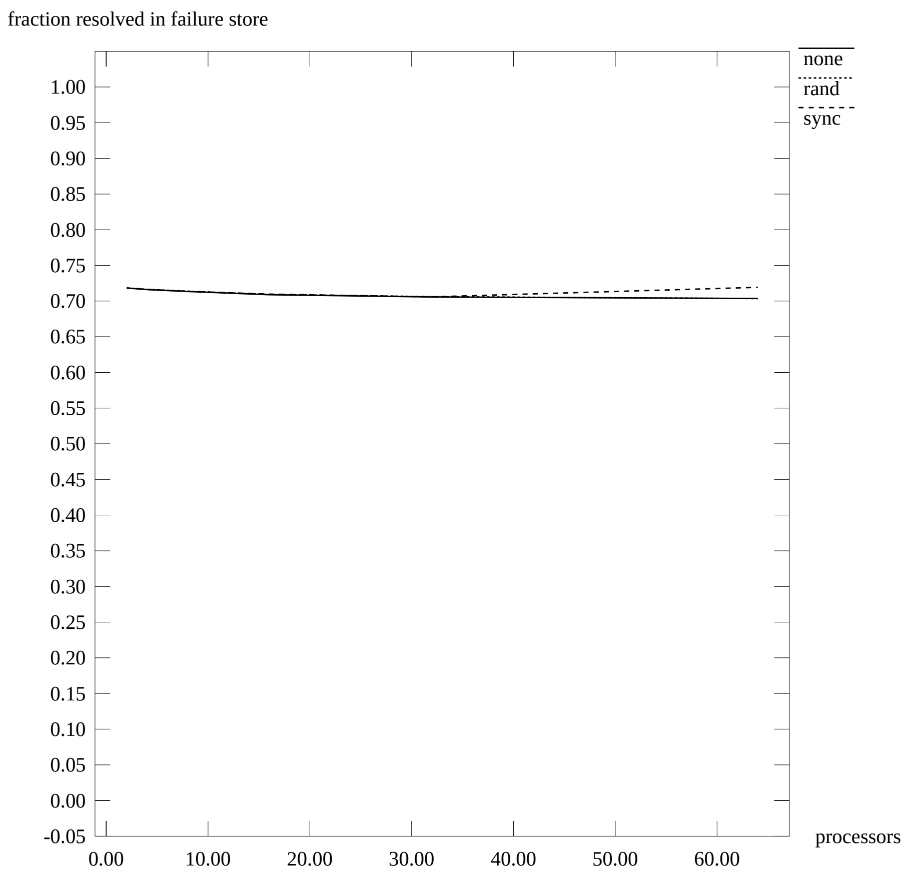


Figure 10: Fraction of subsets resolved in the failure store vs. processors for Input 60:000 .

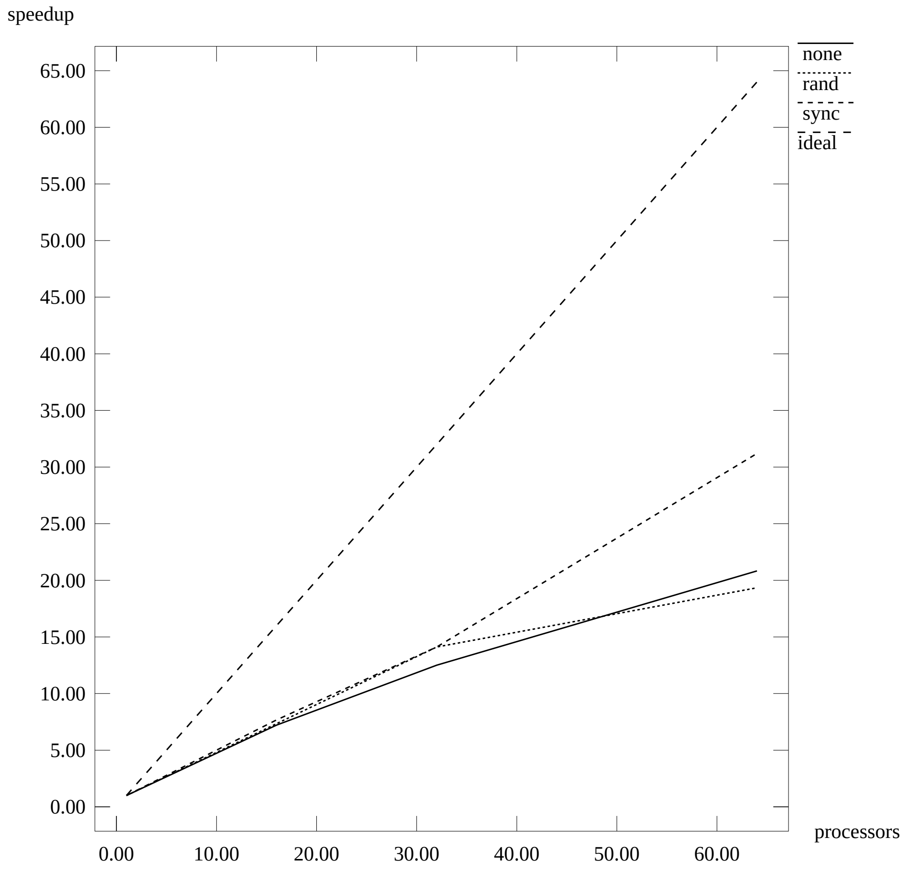


Figure 11: Speedup vs. processors for Input 60:001.

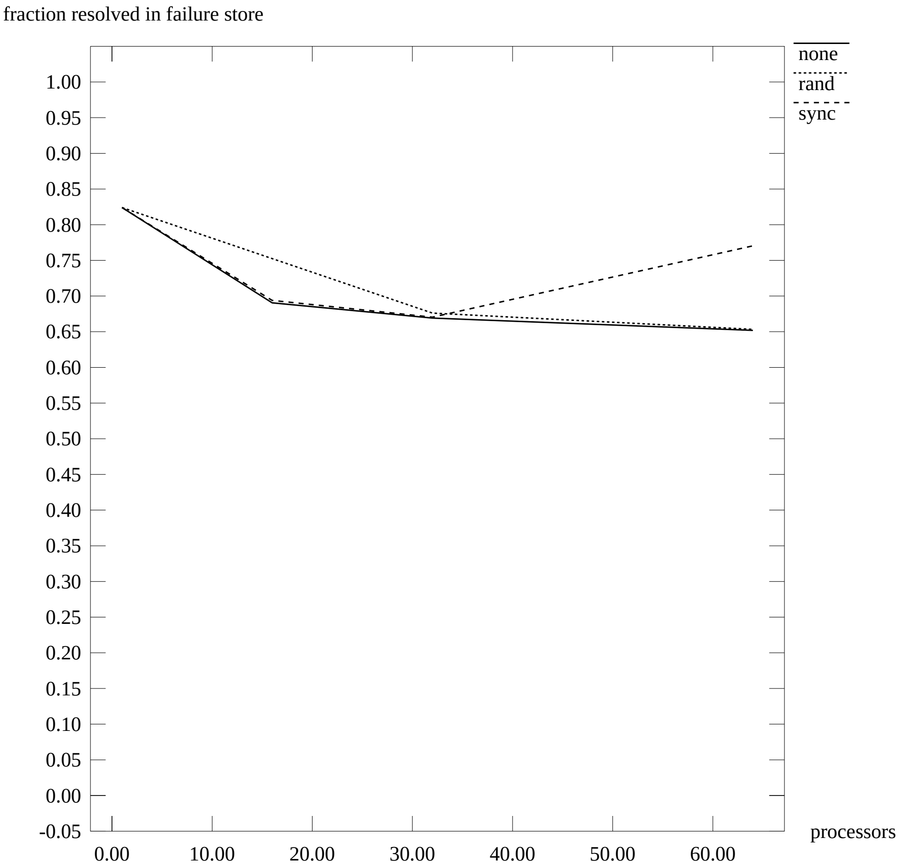


Figure 12: Fraction of subsets resolved in the failure store vs. processors for Input 60:001.

identify general techniques that are useful across similar application. The implementation is, we believe, the fastest implementation of the character compatibility problem. The sequential algorithm for solving the perfect phylogeny problem tasks is due to Agarwala and Fernández-Baca [1], with an improvement suggested by Lawler [9]. Using the full character compatibility application, we were able to solve problems with 60 characters in a few minutes, demonstrating speedups of 50 on 64 processors.

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] E. Brewer and B. Kuszmaul. How to Get Good Performance from the CM-5 Data Network. In *Proceedings of 8th International Parallel Processing Symposium*, Cancun, Mexico, 1994.

[4] Soumen Chakrabarti, Abhiram Ranade, and Katherine Yelick. Randomized load balancing for tree-structured computation. In *Proceedings of the Scalable High Performance Computing Conference*, Knoxville, TN, May 1994.

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

[6] 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.

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

[8] J. Jones. Parallelizing the Phylogeny Problem. Master's thesis., University of California, Berkeley, California, 1994.

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

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

[11] 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.

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

[13] 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.