Phylogenetic Tree Inference on PC Architectures with AxML/PAxML*

Alexandros P. Stamatakis
Technical University of Munich, Department of Computer Science

Thomas Ludwig
Ruprecht-Karls-University, Department of Computer Science

Abstract

Inference of phylogenetic trees comprising hundreds or even thousands of organisms based on the maximum likelihood method is computationally extremely expensive. In previous work, we have introduced Subtree Equality Vectors (SEVs) to significantly reduce the number of required floating point operations during topology evaluation and implemented this method in **(P)AxML**, *which is a derivative of* **(parallel) fastDNAml**. *Experimental results show that* **(P)AxML** *scales particularly well on* inexpensive *PC-processor architectures obtaining global run time accelerations between 51% and 65% over* **(parallel) fastDNAml** *for large data sets, yet rendering* exactly *the same output. In this paper, we present an additional SEV-based algorithmic optimization which scales well on PC processors and leads to a further improvement of global execution times of 14% to 19% compared to the initial version of* **AxML**. *Furthermore, we present novel distance-based heuristics for reducing the number of analyzed tree topologies, which further accelerate the program by 4% up to 8%. Finally, we discuss a novel experimental tree-building algorithm and potential heuristic solutions for inferring large high quality trees, which for some initial tests rendered better trees and accelerated program execution at the same time by a factor greater than 6.*

1. Introduction

Within the ParBaum (Parallel Tree) project at the Technical University of Munich (TUM), work is conducted on parallel phylogenetic tree inference based on the maximum likelihood method by J. Felsenstein [2]. The overall aim of the project is to develop novel systems and algorithms for the computation of huge phylogenetic trees based on sequence data from the ARB [12] database in distributed and parallel environments. In previous work [8, 9, 10] we have introduced Subtree Equality Vectors (SEVs) as a means to reduce topology evaluation time significantly, which represents the by far most cost-intensive part of every phylogenetic tree inference process based on the maximum likelihood method irrespective of the tree building algorithm deployed. We implemented our concept in **(parallel) fastDNAml** [7, 11] and named the resulting program **P**arallel **A(x)**ccelerated **M**aximum **L**ikelihood (**PAxML**). In tests with alignments of 150 up to 500 sequences, we achieved global run time improvements of 26% up to 65% [1] for both the sequential and the parallel version on various platforms. An important result of this work is that the amount of performance improvement primarily depends on the processor architecture and far less on the specific data set used. We found that **(P)AxML** scale particularly well on inexpensive PC processor architectures such as the AMD Athlon MP or Intel Pentium III. E.g. for the same 150 sequence alignment the acceleration achieved with **PAxML** was 26.57% on the Hitachi SR8000-F1 [6] supercomputer and 62.42% on a LINUX cluster. This is due to the fact that SEVs require the execution of a certain amount of integer and pointer arithmetics. Traditional supercomputers have primarily been designed to perform efficiently floating point operations as required for typical supercomputing applications such as fluid dynamics are not well suited for bioinformatics applications, which perform irregular data access in graphs and an important amount of integer arithmetics. Therfore, in this paper, we focus on optimizing and improving our **(P)AxML** code for PC processor architectures and clusters. Furthermore, the HELICS [3] 512 processor PC cluster is at our disposal for computations and PC clusters are inexpensive as well as more adequate for our programs than supercomputers and facilitate installation and adaptation. Finally, **(P)AxML** is a good candidate for a

*This work is sponsored under the project ID **ParBaum**, within the framework of the "Competence Network for Technical, Scientific High Performance Computing in Bavaria": Kompetenznetzwerk für Technisch-Wissenschaftliches Hoch- und Höchstleistungsrechnen in Bayern (KONWIHR). KONWIHR is funded by means of "High-Tech-Offensive Bayern".

[1] The best accelaration values have been achieved with **PAxML**

seti@home-like system [1] due to its coarse-grained parallelism which increases as the tree grows. Such distributed applications primarily run PCs.

The remainder of this paper is organized as follows: In section 2 we describe the SEV method, which is a prerequisite for understanding the novel SEV-based optimization of **(P)AxML** in the same section. This additional optimization rendered performance improvements of 14% to 19% compared to the initial version of **(P)AxML** on PC processor architectures. In section 3 we describe novel distance-based heuristics for further accelerating the program by an average of 4% to 8%. In section 5 we describe results obtained with the modified algorithms. Finally, in section 6 we report about current work covering a new experimental tree building algorithm and potential additional heuristics as well as solutions for the inference of huge phylogenetic trees. First tests with a novel randomized approach render run time improvements of factor > 6 and yield trees with better likelihood scores at the same time. We conclude with a description of future work.

2. Subtree Column Equalities

In general the cost of the likelihood function and the branch length optimization function, which accounts for the greatest portion of execution time (95% in the sequential version of **fastDNAml**), can be reduced in two ways:

Firstly, by reducing the size of the search space using some additional heuristics, i.e. reducing the number of topologies evaluated and thus reducing the number of likelihood function invocations. This approach might, however, overlook high quality trees.

Secondly, by reducing the number of sequence positions taken into account during computation and thus reducing the number of computations at each inner node during each tree's evaluation.

We consider the second possibility through a detailed analysis of column equalities. Two columns in an alignment are equal and belong to the same *column class* if, on a sequence by sequence basis, the base is the same. A homogeneous column consists of the same base, whereas a heterogeneous column consists of different bases.

More formally, let $s_1, ..., s_n$ be the set of aligned input sequences. Let m be the number of sequence positions of the alignment. We say that two columns of the input data set i and j are equal if $\forall s_k, k = 1, ..., n : s_{ki} = s_{kj}$, where s_{kj} is the j-th position of sequence k. One can now calculate the number of equivalent columns for each column class of the input data set.

After calculating column classes, one can compress the input data set by keeping a single representative column for each column class, removing the equivalent columns of the specific class and assigning a count of the number of columns the selected column represents.

Since a necessary prerequisite for a phylogenetic tree calculation is a high-quality multiple alignment of the input sequences, one might expect quite a large number of column equalities on a global level. In fact, this kind of global data compression is already performed by most programs. Unfortunately, as the number of aligned sequences grows, the probability of finding two globally equal columns decreases. However, it is reasonable to expect more equalities on the subtree, or local, level.

The fundamental idea of this paper is to extend this compression mechanism to the subtree level, since a large number of column equalities might be expected on the subtree level. Depending on the size of the subtree, fewer sequences have to be compared for column equality and thus, the probability of finding equal columns is higher.

None the less, we restrain the analysis of subtree column equality to homogeneous columns for the following reason:

The calculation of heterogeneous equality vectors at an inner node p is complex and requires the search for c^k different column equality classes, where k is the number of tips (sequences) in the subtree of p and c is the number of distinct values the characters of the sequence alignment are mapped to (e.g. **fastDNAml** uses 15 different values). This overhead would not amortize well over the additional column equalities we would obtain, especially when $c^k > m'$.

We now describe an efficient and easy way for recursively calculating subtree column equalities using Subtree Equality Vectors (SEVs).

Let s be the virtual root placed in an unrooted tree for the calculation of its likelihood value. Let p be the root of a subtree with children q and r, relative to s. Let ev_p (ev_q, ev_r) be the equality vector of p (q, r, respectively), with size m', where m' is the length of the compressed global sequences. The value of the equality vector for node p at position i, where $i = 1, ..., m'$ can be calculated by the following function:

$$ev\_p(i) = \begin{cases} ev\_q(i) & if \quad ev\_q(i) = ev\_r(i) \\ -1 & else \end{cases} \tag{1}$$

If p is a leaf, we set $ev\_p(i) := map(sequence\_p(i))$, where, $map()$ is a function that maps the character representation of the aligned input sequence $sequence\_p$ at leaf p to values $0, 1, ..., c$. Thus, the values of an inner SEV ev_p, at position i, range from $-1, 0, ..., c$, i.e. -1 if column i is heterogeneous and from $0, ..., c$ in the case of an homogeneous column. For SEV values $0, ..., c$ a pointer array $ref\_p(c)$ is maintained, which is initialized with $NULL$ pointers, for storing the references to the first occurrence of the respective column equality class in the likelihood vector of the current node p.

Thus, if the value of the equality vector $ev\_p(j) > -1$ and $ref\_p(ev\_p(j)) \neq NULL$ for an index j of the likelihood vector $lv\_p(j)$ of p, the value for the specific homogeneous column equality class $ev\_p(j)$ has already been calculated for an index $i < j$ and a large block of floating point operations can be replaced by a simple value assignment $lv\_p(j) := lv\_p(i)$. If $ev\_p(j) > -1$ and $ref\_p(ev\_p(j)) = NULL$, we assign $ref\_p(ev\_p(j))$ to the address of $lv\_p(j)$, i.e. $ref\_p(ev\_p(j)) := adr(lv\_p(j))$.

The additional memory required for equality vectors is $O(n * m')$. The additional time required for calculating the equality vectors is $O(m')$ at every node.

The initial approach renders global run time improvements of 12% to 15%. These result from an acceleration of the likelihood evaluation function between 19% and 22%, which in turn is achieved by a reduction in the number of floating point operations between 23% and 26% in the specific function.

It is important to note that the initial optimization is only applicable to the likelihood evaluation function, and *not* to the branch length optimization function. This limitation is due to the fact that the SEV calculated for the *virtual* root placed into the topology under evaluation, at either end of the branch being optimized, is very sparse, i.e. has few entries > -1. Therefore, the additional overhead induced by SEV calculation does not amortize well with the relatively small reduction in the number of floating point operations (2% - 7%). Note however, that the SEVs of the *real* nodes at either end of the specific branch do not need to be sparse, this depends on the number of tips in the respective subtrees.

We now show how to efficiently exploit the information provided by an SEV in order to achieve an additional reduction in the number of floating point operations by extending this mechanism to the branch length optimization function.

In order to make better use of the information provided by an SEV at an inner node p with children r and q, it is sufficient to analyze at a high level how a single entry i of the likelihood vector at p, $lv\_p(i)$ is calculated:

$$lv\_p(i) = f(g(lv\_q(i), z(p,q)), g(lv\_r(i), z(p,r))) \quad (2)$$

where $z(p,q)$ ($z(p,r)$) is the length of the branch from p to q (p to r respectively). Function $g()$ is a computationally expensive function that calculates the likelihood of the left and the right branch of p respectively, depending on the branch lengths and the values of $lv\_q(i)$ and $lv\_r(i)$, whereas $f()$ performs some simple arithmetic operations for combining the results of $g(lv\_q(i), z(p,q))$ and $g(lv\_r(i), z(p,r))$ into the value of $lv\_p(i)$. Note that $z(p,q)$ and $z(p,r)$ do not change with i.

If we have $ev\_q(i) > -1$ and $ev\_q(i) = ev\_q(j)$, $i < j$, we have $lv\_q(i) = lv\_q(j)$ and therefore $g(lv\_q(i), z(p,q)) = g(lv\_q(j), z(p,q))$ (the same equality holds for node r). Thus, for any node q we can avoid the recalculation of $g(lv\_q(i), z(p,q))$ for all $j > i$, where $ev\_q(j) = ev\_q(i) > -1$. We precalculate those values and store them in arrays $precalc\_q(c)$ and $precalc\_r(c)$ respectively, where c is the number of distinct character-value mappings found in the sequence alignment.

Our final optimization consists in the elimination of value assignments of type $lv\_q(i) := lv\_q(j)$, for $ev\_q(i) = ev\_q(j) > -1$, $i < j$ where i is the first entry for a specific homogeneous equality class $ev\_q(i) = 0, ..., c$ in ev_q. We need not assign those values due to the fact that $lv\_q(j)$ will never be accessed. Instead, since $ev\_q(j) = ev\_q(i) > -1$ and the value of $g\_q(j) = g\_q(i)$ has been precalculated and stored in $precalc\_q(ev\_p(i))$, we access $lv\_q(i)$ through its reference in $ref\_q(ev\_q(i))$.

During the main for-loop in the calculation of lv_p we have to consider 6 cases, depending on the values of ev_q and ev_r. For simplicity we will write $p\_q(i)$ instead of $precalc\_q(i)$ and $g\_q(i)$ instead of $g(lv\_q(i), z(p,q))$.

$$lv\_p(i) := \begin{cases} f(p\_q(ev\_q(i)), p\_r(ev\_r(i))) \\ \textbf{if} ev\_q(i) = ev\_r(i) > -1, \\ ref\_p(ev\_r(i)) = NULL \\ \\ skip \\ \textbf{if} ev\_q(i) = ev\_r(i) > -1, \\ ref\_p(ev\_r(i)) \neq NULL \\ \\ f(p\_q(ev\_q(i)), p\_r(ev\_r(i))) \\ \textbf{if} ev\_q(i) \neq ev\_r(i), \\ ev\_q(i), ev\_r(i) > -1 \\ \\ f(p\_q(ev\_q(i)), g\_r(i)) \\ \textbf{if} ev\_q(i) > -1, ev\_r(i) = -1 \\ \\ f(g\_q(i), p\_r(ev\_r(i))) \\ \textbf{if} ev\_r(i) > -1, ev\_q(i) = -1 \\ \\ f(g\_q(i), g\_r(i)) \\ \textbf{if} ev\_q(i) = -1, ev\_r(i) = -1 \end{cases} \quad (3)$$

For a more thorough description of SEVs see [10].

Additional Algorithmic Optimization: Since the initial implementation was designed for no particular target platform and **(P)AxML** scales best on PC processor architectures, we investigated additional algorithmic optimizations especially designed for these architectures. An additional acceleration can be achieved by a more thorough exploitation of SEV information in function `makenewz()`, which optimizes the length of a *specific* branch b and accounts for approximately one third of total execution time. Function `makenewz()` consists of two main parts: Initially, a for-loop over all alignment positions is executed for computing the likelihood vector of the virtual root s placed into branch b connecting nodes p and q. Thereafter, a do-loop is executed which iteratively alters the branch length according to a convergence criterion. For calculating the new likelihood value of the tree for the altered branch length within that do-loop, an inner for-loop over the likelihood vector of the virtual root s which uses the data computed by the initial for-loop is executed.

A detailed analysis of `makenewz()` reveals two points for further optimization:

Firstly, the do-loop for optimizing branch lengths is rarely executed more than once (see Table 1). Furthermore, the inner for-loop accesses the data computed by the initial for-loop. Therefore, we integrated the computations performed by the *first* execution of the inner for-loop into the initial for-loop and appended the conditional statement which terminates the iterative optimization process to the initial for-loop, such as to avoid the computation of the *first* inner for-loop completely.

# seq.	# invoc.	# invoc. $it > 1$	avg $it > 1$
10	1629	132	7.23
20	8571	661	6.14
30	21171	1584	6.17
40	39654	2909	6.21
50	63112	4637	6.26

Table 1. makenewz() analysis

Secondly, when more than one iteration is required for optimizing the branch length in the do-loop we can reduce the length of the inner for-loop by using SEVs. The length of the inner for-loop $f = m'$ can be reduced by $nn - c$ the number of non-negative entries nn of the SEV at the virtual root s minus the number c of distinct column equality classes, since we need to calculate only one representative entry for each column equality class. Note that the weight of the column equality class representative is the accumulated weight of all column equalities of the specific class at s. Thus, the reduced length f' of the inner for-loop is obtained by $f' := m' - nn + c$.

We obtain the SEV ev_s of the virtual root s by applying:

$$ev\_s(i) := \begin{cases} ev\_p(i) & if \quad ev\_p(i) = ev\_q(i) \\ -1 & else \end{cases} \tag{4}$$

Since the branch length optimization process requires a sufficiently large average number of iterations to converge if it does not converge after the *first* iteration (see Table 1) our optimization scales well despite the fact that the SEV at the virtual root s is relatively sparse, i.e. $nn - c$ is relatively small compared to m'.

3. Distance-based Heuristics

Although the maximum-likelihood method is *not* distance-based, sequence distance has some impact on the inference process, especially when inferring large trees with organisms from all three kingdoms (Eucarya, Bacteria, Archaea).

Thus, we implemented and tested simple distance-based heuristics, which enable skipping the evaluation of a certain amount of topologies. Let us consider the tree-building algorithm of **(P)AxML**. Suppose we have already calculated the best tree t_k containing the first k sequences of the sequence input order. Tree t_{k+1} is obtained by inserting sequence $k + 1$ into all $2k - 3$ branches of t_k and evaluating the respective topologies. The algorithm starts with the only possible tree topology for 3 sequences, i.e. with t_3. Note that k branches of t_k lead to a leaf (sequence) which we will call "terminal branches" from now on. If sequence $k + 1$ finally is inserted into such a terminal branch, it has to be closely related to the sequence at this terminal branch in terms of sequence distance. We call topologies constructed by insertion of sequence $k + 1$ into a terminal branch, "terminal branch topologies". For implementing distance-based heuristics before each transition $k \to k + 1$ we calculate a score vector $score_{k+1}$ of size k for each terminal branch topology by applying a simple score-function (number of equal characters) to the sequence pairs $(1, k + 1), ..., (k, k+1)$. The amount of terminal branch topologies to be skipped is determined by calculating the difference $\Delta_{k+1} = max(score_{k+1}) - min(score_{k+1})$ between the lowest and the highest score of $score_{k+1}$ and comparing it to the average avg_k of $\Delta_4, ..., \Delta_k$ of all previous insertions $4, ..., k$. If $\Delta_{k+1} < avg_k$ we do not skip any topologies. This is done to avoid skipping terminal branch topologies generated by insertion of a sequence that fits equally bad or good into all terminal branches. If $\Delta_{k+1} > avg_k$ we skip a portion $min(0.8, 1 - (\Delta_{k+1}/avg_k))$ of the worst scores in $score_{k+1}$ (0.8 proved to be a good value in our experiments leading to few deviations in the final tree, see section 5).

Note that this method can equivalently be used with the local and/or global rearrangement option where after each transition $k \to k + 1$ the tree t_{k+1} is rearranged for further improving its likelihood. This is due to the fact that rearranging the tree will once again yield a certain amount of terminal branch topologies.

4. Implementation

We implemented the new concepts described in sections 2 and 3 in the original version of the sequential program **AxML (v1.7)**. We name the new version containing the algorithmic optimizations **AxML (v2.5)** and the program which in addition contains the distance-based heuristics **AxML (v3.0)**. The new versions of our parallel code are named **PAxML (v1.5)** and **PAxML (v1.6)** respectively.

Furthermore, we integrated a random sequence input order permutation generator for testing the distance-based heuristics. **AxML (v3.0)** loops over random permutations and invokes the de novo tree building function twice for each permutation (with and without heuristics) for comparing the resulting trees and execution times. This modification was carried out to enable thorough testing and for evaluating the impact of the sequence input order on distance-based heuristics.

Finally, the alternative tree building algorithm described in section 6 has been implemented in **AxML (v4.0)**, which can optionally be executed with distance-based heuristics.

AxML and **PAxML** are freely available for download at: `wwwbode.in.tum.de/˜stamatak/research.html`.

5. Results

Platforms & Test Data: For testing the new **(P)AxML** programs, we used a small LINUX cluster equipped with 16 Intel Pentium III processors and 1 Gbyte of memory and the HELICS cluster with 512 AMD Athlon MP processors and 2 Gbyte of memory. Both clusters are interconnected by Myrinet. We extracted several alignments comprising organisms from all three kingdoms (Eucarya, Bacteria, Archaea) from the small subunit ribosomal RiboNucleic Acid (ssrRNA) database of the ARB system [12] and also used a 56 sequence alignment provided as test set with the **parallel fastDNAml** distribution [4].

Results: In Table 2 we list the global run time (secs) of **AxML (v2.5)**, **AxML (v1.7)** and **fastDNAml (v1.2.2)** for alignments containing 150, 200, 250 and 500 sequences. The tree inference was conducted without global and local rearrangements (for details on program options see fastDNAml documentation and [7]).

# sequences	v2.5	v1.7	fastDNAml
150	632	748	1603
200	1227	1443	3186
250	2055	2403	5431
500	10476	12861	26270

Table 2. AxML (v1.7)/(v2.5) vs. fastDNAml

In Table 3 we describe results obtained by comparing the output of the de novo tree building function with and without distance-based heuristics for sets of randomized sequence input order permutations. All tests in the first four rows were executed without local and/or global rearrangements, whereas the test with the 56 sequence alignment was conducted with rearrangements (local and global rearrangements set to 1) to prove that our distance-based heuristics work equally well with this program option. Finally, the last line of this table 150(P,R) refers to the global execution times of one parallel run with **PAxML (v1.5)** and **PAxML (v1.6)** respectively which were executed with local and global rearrangements (local and global rearrangements set to 1). Column *patterns* gives the number of distinct

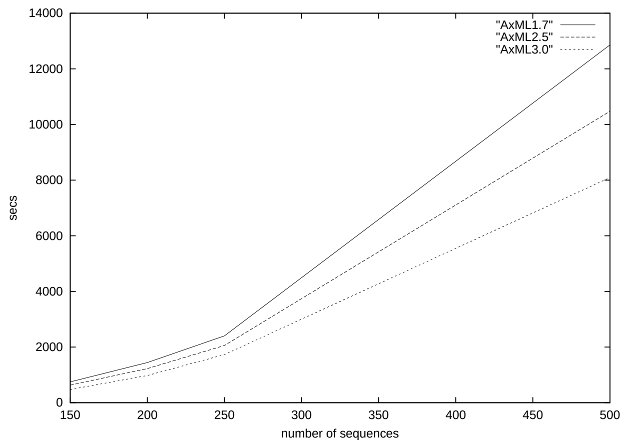

Figure 1. Execution times of AxML versions

patterns in the alignments, column *permutations* indicates the number of randomized input order permutations evaluated. Furthermore, *skip* represents the number of topologies skipped by our heuristics and *topol* the total number of generated topologies. In addition, we measured the average and maximum (*improvement* and *max improvement*) run time improvement for the de novo tree building function and counted the number of differing trees. Finally, we calculated the maximum deviation in final likelihood values (*max diff*). Table 3 indicates that our heuristics are sound, i.e. only a few final trees differ and are thus well-suited especially for quickly evaluating a large number of sequence input order permutations and calculating an initial set of good trees which can then be refined in a second step without heuristics (see section 6). We note however that the *performance* of the heuristics depends on the sequence input order due to significant differences in measured average and maximum run time improvements. Finally Table 3 indicates that the average run time improvement achieved by distance-based heuristics increases with tree size. In Figure 1 we depict the global run times for all **AxML** versions for one specific input order permutation of the 150, 200, 250 and 500 sequences alignments on a Pentium III processor.

# sequences	patterns	permutations	skip	topol	improvement	max improvement	diff	max diff
150	2137	569	1237.5	21904	5.63%	20.43%	4	-0.26%
200	2253	222	2297.3	39204	5.73%	20.04%	3	-0.02%
250	2330	133	4263.4	61504	6.89%	15.99%	4	-0.03%
500	2751	28	17266.9	248004	7.84%	20.72%	3	-0.04%
56(R)	386	48	358.0	8274	4.26%	18.86%	1	-0.05%
150(P,R)	2137	1	9476	50948	22.75%	22.75%	0	0.00%

Table 3. Tree quality and inference times with and without heuristics

6. Current & Future Work

In this section we present experimental work and discuss potential solutions and novel approaches for the inference of huge trees.

An Alternative Algorithm: As already mentioned, the input order of the sequences has a major impact on the likelihood of the final tree.

In order to investigate this problem further, we implemented an alternative tree building algorithm, which is currently at an experimental stage. We call the respective program **AxML (v4.0)**.

As already described in section 3, the tree building algorithm of **AxML (v2.5)** progressively inserts the sequences into the tree. Our consideration is that the dependency on sequence input order may eventually be reduced by changing the algorithm at the transition $t_k \rightarrow t_{k+1}$ as follows: Let t_k be the best tree comprising k sequences (not necessarily the first k sequences of the input order, see below). Instead of inserting only sequence $k+1$ into all branches of t_k and evaluating the respective topologies, we insert all remaining $n-k-1$ sequences from $1,...,n$, that have not already been inserted into the tree, into all branches of t_k and continue with the best such obtained tree t_{k+1}. Finally, we mark the sequence which was added to the tree at transition $k \rightarrow k+1$ as inserted.

Initial tests with small sequence alignments (40, 50 and 56 taxa) of relatively closely related organisms, and a large 150 sequence alignment comprising organisms of all three kingdoms (Eucarya, Bacteria, Archaea) suggested that **AxML (v4.0)** might render better results than the standard program version for closely related organisms, whereas it does not seem to perform well for alignments containing distant organisms. For an initial test of this hypothesis we extracted three alignments comprising 100 closely related organisms each from the ARB database and executed test runs with **AxML (v4.0)** and **AxML (v2.5)**.

In Table 4 we depict the execution times, global and local rearrangement settings, as well as the final Ln likelihood values for **AxML (v2.5)** and **AxML (v4.0)** on a Sun-Blade-1000.

Initial results suggest that the use of **AxML (v4.0)** might be a good option for building maximum likelihood trees for closely related organisms, since in some cases it yields better results than **AxML (v2.5)** at a significantly lower computational cost, as indicated in Table 4.

Note that the changes introduced in **AxML (v4.0)** can easily be integrated into **PAxML** since only few lines of additional code are required for adapting the tree building algorithm.

Solutions for Huge Trees: The inference of large trees is limited by two main factors.

Firstly, in order to improve the quality of the final tree a high amount of local, regional and global rearrangements should be performed for avoiding local maxima. On the other hand this practice significantly increases the computational cost. E.g. the sequential inference of a 150 taxa tree can be carried out within a few *minutes* without local and/or global rearrangements, whereas with local and global rearrangements only set to 1 it requires several *hours*.

Secondly, in order to attain some kind of confidence that the final tree is close to the best tree the inference process should be repeatedly executed with several (randomized) sequence input order permutations.

Thus, although we have been able to infer a 1000 taxa tree on HELICS (global and local rearrangements set to 1), the acceleration achieved over **parallel fastDNAml** by using SEVs is not sufficient for inferring huge trees of high quality.

In this section we discuss various possibilities to further accelerate the program and handle the problems mentioned above.

During the execution of **AxML (v3.0)** with randomized sequence input order permutations and global as well as local rearrangements switched off we observed that tree values close to, or even better than those originally obtained by the execution of **PAxML** (global and local rearrangements set to 1) with the default sequence input order were achieved with some random permutations.

Such a randomized approach has several advantages, since the distributed computation of a great number of input sequence order permutations with **AxML** can be performed

version	data set	execution time (secs)	local	global	Ln likelihood	heuristics used?	difference
v2.5	100_1	7922.39	1	1	-25797.84	No	-0.49%
v2.5	100_1	50390.96	2	2	-25669.88	No	best
v4.0	100_1	8517.61	1	1	-25691.23	No	-0.08%
v4.0	100_1	8308.54	1	1	-25691.23	Yes	-0.08%
v2.5	100_2	7136.94	1	1	-25429.59	No	-0.90%
v2.5	100_2	43826.03	2	2	-25381.05	No	-0.72%
v4.0	100_2	7922.35	1	1	-25198.63	No	best
v4.0	100_2	7456.53	1	1	-25198.63	Yes	best
v2.5	100_3	5452.92	1	1	-23885.10	No	-0.13%
v2.5	100_3	34406.41	2	2	-23852.59	No	best
v4.0	100_3	6813.99	1	1	-23918.41	No	-0.27%
v4.0	100_3	6461.43	1	1	-23892.74	Yes	-0.16%

Table 4. Comparison of AxML (v2.5) and (v4.0) for three alignments of closely-related sequences

with a seti@home-like distributed system due to its coarse-grained parallelism. Furthermore, the best trees obtained by such a randomized tree inference can be used to build a consensus tree with CONSENSE [5]. Such a consensus tree or a certain portion of the best trees can then be globally rearranged on a parallel computer using **PAxML**. Finally, the permutation(s) which rendered the best tree(s) can be used for a large parallel or distributed run with **PAxML** with high rearrangement levels.

In Table 5 we summarize first results for such a randomized approach which were obtained by an appropriately modified version of **AxML (v3.0)** on the small 16-node LINUX cluster. For each data set we generated a set of random sequence input order permutations (*perm*) and inferred the respective tree without local and global rearrangements. Thereafter, we rearranged only the best final tree for each data set globally. We measured total required CPU hours for the randomized approach and CPU hours for *one* complete parallel run with **PAxML** and the default input sequence ordering for determining the acceleration factor *acc*. Columns *L* and *G* indicate the respective setting of the rearrangement option for **PAxML** and *RR* indicates the global rearrangement setting for the best randomized tree. In *all* cases we achieved a slightly better final likelihood with the randomized approach (*impr*) at a significantly lower computational cost (see *acc*).

Thus, our randomized approach significantly reduces the computational cost for the inference of large phylogenetic trees and provides a framework for handling sequence input order permutations. Furthermore, it produces trees with slightly improved likelihood values and at the same time provides a set of several good trees and permutations which can then be used for one large parallel run. Finally, our approach represents a significantly faster alternative to the recommended practice of executing **parallel fastDNAml/PAxML** repeatedly with random input order permutations and a high level of global and local rearrangements which is a substantial limiting factor for the inference of large phylogenetic trees due to its high computational cost.

Another important observation within this context is that the number m' of distinct column patterns in the alignment has an impact on program performance. Since the complexity of **(parallel) fastDNAml** is linear in m' we can achieve a linear acceleration of the program by reducing the number of patterns by some heuristic criterion to $m'' < m'$. We call such heuristics "pattern reduction methods". Note however that pattern reduction might not scale equally well to **(P)AxML** if patterns containing a large number of equal characters are eliminated, since this reduces the amount of subtree column equalities. An additional advantage of pattern reduction is that it reduces the memory requirements of **(P)AxML** which are approximately $O(n * m')$, i.e. the size of the sequence alignment and may become a bottleneck for huge trees.

# seq	# perm	RR	L	G	acc	impr.
150	50	1	1	1	6.17	+0.04%
150	50	2	5	5	15.68	+0.00%
200	29	1	1	1	6.21	+0.08%
250	16	1	1	1	9.60	+0.10%

Table 5. Results for randomized tree inference

The aptness of a pattern reduction method can be evaluated in various ways. Initially the likelihood $lh(m'', m')$ of the final tree $t(m'')$ obtained by the reduced pattern m'' is recomputed with the full pattern m' and compared with the likelihood $lh(m')$ obtained by a de novo computation with the full pattern for the same input order permutation. We repeat this process for several data sets and randomized input order permutations.

If there are little or no deviations between $lh(m'', m')$ and $lh(m')$ we can use the reduced pattern for tree reconstruction.

If pattern reduction shall be used for an initial fast evaluation of a great number of input order permutations we can analyze the correlation between the likelihood values of the sets $t(m'', i)$, $i = 1, ..., r$ with the reduced pattern and $t(m', i)$ with the complete pattern. If likelihood values of $t(m'', i)$ and $t(m', i)$ are correlated, especially among the trees with the best likelihoods we can use the criterion for a fast evaluation of sequence input order permutations.

We consider the first point to be appropriate for the evaluation of more conservative heuristics which reduce the length of m' by a small portion only, whereas the second point can be used for the evaluation of more aggressive heuristics.

One good conservative criterion is to skip all those columns consisting mainly of gaps and only a small number of characters, e.g. 4 or 5 and has already been implemented in an experimental version of **AxML**. Applying this criterion already yields a significantly smaller number of patterns for large alignments e.g. 1915 instead of 2137 for the 150 taxa test set (10.39% pattern reduction). The full evaluation of that tree, i.e. with all patterns rendered *exactly* the same likelihood.

Furthermore, this approach has already been used for computing trees for randomized sequence input order permutations for the 150, 200 and 250 taxa trees mentioned in Table 5.

An approach for more aggressive heuristics is to reduce the number of patterns by performing one or several initial evaluations with all patterns and analyzing the contribution of each pattern to the intermediate and final likelihood values. Note that the contribution of each pattern to the final likelihood value appears to be relatively invariant for a set of random input order permutations.

Then some simple criterion for eliminating patterns can be applied, e.g. skipping a certain percentage of the worst patterns, i.e. those patterns that contribute little.

Future Work: We are going to further investigate the applicability of pattern reduction methods and analyze if good permutations have intrinsic properties.

Our main focus is going to be on building a large distributed seti@home-like system for phylogenetic tree inference. We are planning to implement a flexible client, able to provide randomized sequential tree inference services optionally including distance-based or pattern reduction heuristics or an evaluation with **ATrExML** [10, 13], as well as parallel style topology evaluation services, similar to the worker component in **PAxML**. We are planning to split up the inference process into two phases. Initially, a large set of randomized input order permutations will be evaluated by the clients and stored by the main server. In the second phase the best randomized trees will be globally rearranged and the best input order permutations will be used for de novo tree calculations in a parallel style. We believe that such an approach will provide the potential for building large trusted trees of 1000 taxa and more.

Acknowledgments: We would like to thank Markus P¨ogel for giving us access to the LINUX cluster and Dr. Harald Meier for providing us the alignments from the ARB database.

References

[1] Berkley. Setiathome homepage. Technical report, SETIATHOME.SSL.BERKELEY.EDU, 2002.

[2] J. Felsenstein. Evolutionary trees from dna sequences: A maximum likelihood approach. *J. Mol. Evol.*, 17:368–376, 1981.

[3] HeLiCs. Heidelberg linux cluster. Technical report, HELICS.UNI-HD.DE, 2002.

[4] Indiana-State-University. Parallel fastdnaml. Technical report, WWW.INDIANA.EDU/ RAC/HPC/FASTDNAML, 2001.

[5] L. Jermiin, G. Olsen, K. Mengersen, and S. Easteal. Majority-rule consensus of phylogenetic trees obtained by maximum-likelihood analysis. *Mol. Biol. Evol.*, 14:1297–1302, 1997.

[6] LRZ. The hitachi sr8000-f1. Technical report, WWW.LRZ-MUENCHEN.DE/SERVICES/COMPUTE/HLRB, 2002.

[7] G. Olsen, H. Matsuda, R. Hagstrom, and R. Overbeek. fastdnaml: A tool for construction of phylogenetic trees of dna sequences using maximum likelihood. *Comput. Appl. Biosci.*, 10:41–48, 1994.

[8] A. P. Stamatakis, T. Ludwig, and H. Meier. Adapting paxml to the hitachi sr8000-f1 supercomputer. In *Proceedings of 1. Joint HLRB and KONWIHR Workshop*, October 2002.

[9] A. P. Stamatakis, T. Ludwig, H. Meier, and M. J. Wolf. Accelerating parallel maximum likelihood-based phylogenetic tree computations using subtree equality vectors. In *Proceedings of SC2002*, November 2002.

[10] A. P. Stamatakis, T. Ludwig, H. Meier, and M. J. Wolf. Axml: A fast program for sequential and parallel phylogenetic tree calculations based on the maximum likelihood method. In *Proceedings of CSB2002*, August 2002.

[11] C. Stewart, D. Hart, D. Berry, G. Olsen, E. Wernert, and W. Fischer. Parallel implementation and performance of fastdnaml - a program for maximum likelihood phylogenetic inference. In *Proceedings of SC2001*, November 2001.

[12] TUM. The arb project. Technical report, WWW.ARB-HOME.DE, 2002.

[13] M. Wolf, S. Easteal, M. Kahn, B. McKay, and L. Jermiin. Trexml: A maximum likelihood program for extensive tree-space exploration. *Bioinformatics*, 16(4):383–394, 2000.