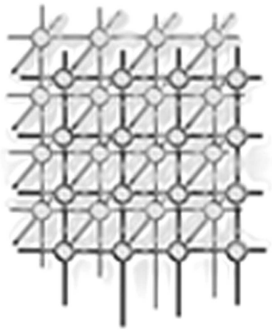
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2003; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*



The AxML program family for maximum likelihood-based phylogenetic tree inferrence

Alexandros P. Stamatakis[1,*,*] , Thomas Ludwig[2], and Harald Meier[1]

[1] *Technische Universität München, Lehrstuhl für Rechnertechnik und Rechnerorganisation/I10, Boltzmannstr. 3 D-85748 Garching b. München, Germany*
[2] *Ruprecht-Karls-Universität Heidelberg, Institut für Informatik, Im Neuenheimer Feld 348, D-69120 Heidelberg, Germany*

SUMMARY

Inference of phylogenetic (evolutionary) trees comprising hundreds or thousands of organisms based on the maximum likelihood criterion is a computationally extremely intensive task. This paper describes the evolution of the AxML program family which provides novel algorithmic as well as technical solutions for the maximum likelihood-based inference of huge phylogenetic trees. Algorithmic optimizations and a new tree building algorithm yield run time improvements of a factor > 4 compared to fastDNAml and parallel fastDNAml returning equally good trees at the same time. Various parallel, distributed, and grid-based implementations of AxML provide the program the capability to acquire the large amount of required computational resources for the inference of huge high quality trees.

KEY WORDS: Phylogenetic trees; maximum likelihood; parallel and distributed computing

*Correspondence to: Technische Universität München, Lehrstuhl für Rechnertechnik und Rechnerorganisation/I10, Boltzmannstr. 3 D-85748 Garching b. München, Germany
*E-mail: stamatak@cs.tum.edu
Contract/grant sponsor: 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". All major parallel tests have been carried out on the HEidelberg LInux Cluster System (HELICS).

Received 31 January 2000
Revised 19 September 2002

1. INTRODUCTION

In recent years there has been an astonishing accumulation of genetic information for many different organisms. This information can be used to infer evolutionary relationships (called a *phylogenetic tree* or *phylogeny*) among a collection of species or even closely related subspecies. There are a variety of techniques that are used to compute these relationships, including the use of maximum likelihood. A recent result by Korber *et al.* that times the evolution of the HIV-1 virus [9] demonstrates that maximum likelihood techniques can be effective in solving biological problems.

Maximum likelihood approaches start with a collection of taxa and a (binary) tree representing possible relationships. Each taxa is represented by a nucleotide or amino acid sequence denoted by characters. The sequences from the individual taxa are aligned and then on a column-by-column basis under certain evolutionary assumptions the likelihood of each column is computed. The overall likelihood is a function of all the column likelihoods. Typically, maximum likelihood programs generate a variety of trees to determine the most likely tree as well as other good trees that are not statistically significantly different from the most likely tree. Because of computational requirements of likelihood analysis and the large number of possible trees, relatively few trees are ever considered by maximum likelihood approaches, especially for large numbers of taxa.

Some earlier work in this area of genome analysis focused on finding *perfect phylogenies*. Kannan and Warnow have a polynomial time algorithm for finding perfect phylogenies [8] under certain reasonable restrictions. However, like many problems associated with genome analysis, the general version of the perfect phylogeny problem is NP-complete [1]. Perfect phylogenies require that for each character in each column, the taxa containing that character in that column form a subtree of the phylogeny. While maximum likelihood methods do not strive to meet this requirement (and regularly produce highly likely, yet "imperfect" trees), it is widely believed that computing phylogenies that meet any sort of effective criteria is NP-hard. Thus, the introduction of heuristics for reducing the search space in terms of potential tree topologies evaluated becomes inevitable. Heuristics for phylogenetic tree calculations still remain computationally expensive, mainly due to the high cost of the tree likelihood function, which is invoked repeatedly for each tree topology analyzed.

Thus, only relatively small trees of high quality (150 [21], 228 [2] taxa), have been calculated so far, although large data sets containing potential phylogenetic information are available (e.g. approximately 30000 sequences in the ARB [11] ssu rRNA database).

Therefore, technical solutions for obtaining the required amount of computational resources as well as algorithmic solutions for improving heuristics and accelerating the evaluation of the likelihood function are required.

In this article we present the algorithmic and technical evolution of the **AxML** (**Ax**elerated **M**aximum **L**ikelihood) program family (see Figure 1) with emphasis on algorithmic results. The **AxML** program family has been derived from **fastDNAml** [12] and **parallel fastDNAml** [21] [22] respectively.

In Section 2 we describe algorithmic optimizations of the topology evaluation function based on Subtree Equality Vectors (SEVs) [16] [17] and the design of a new partially randomized algorithm [19] which further accelerates the tree building process. In Section 3 we briefly

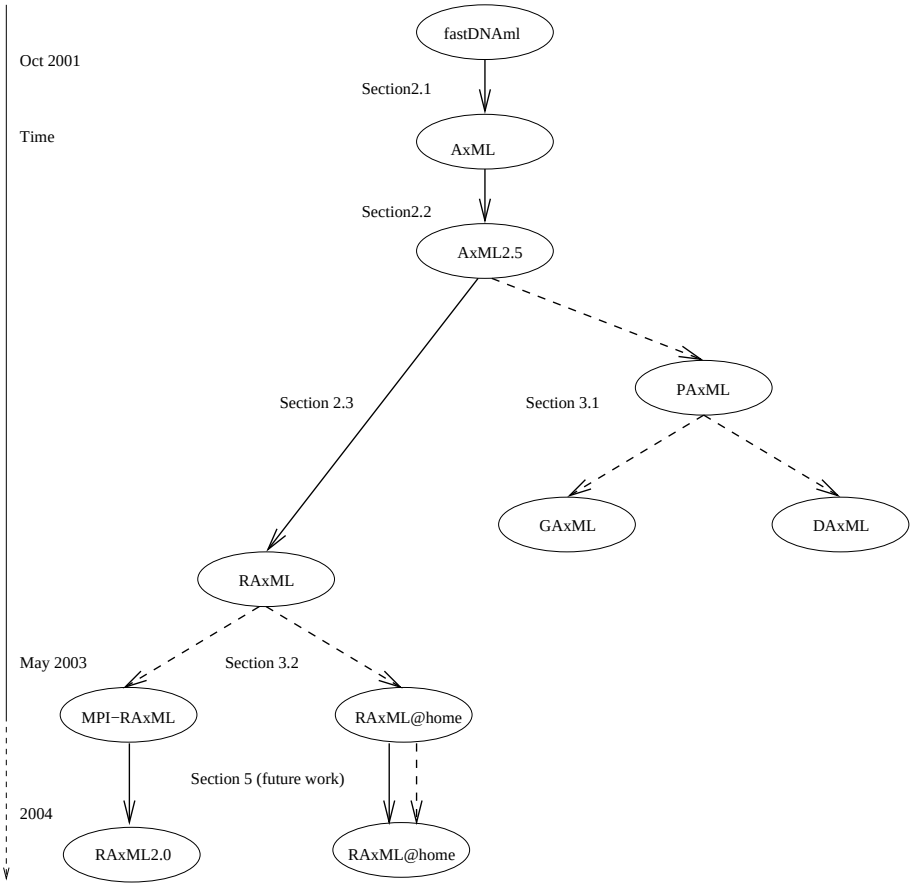


Figure 1. Evolutionary tree of the **AxML** program family; dotted lines indicate technical evolution, straight lines indicate algorithmic evolution

outline some Grid- and CORBA-based [20] technical solutions for **AxML**. Finally, we briefly summarize the most important results and mention aspects of current and future work.

1.1. Related work

For reducing the size of the search space heuristics, such as e.g. the stepwise addition algorithm (introduced in [5], modified in [12]), the advanced stepwise addition algorithm [27] or quartet puzzling [23], are required for the tree building process.

Quartet puzzling yields trees of comparable quality but is slower than stepwise addition and thus not well suited for reconstruction of big trees. One of the main shortfalls of the stepwise

addition algorithm as implemented in **fastDNAml** and initial versions of **AxML** is that the final tree strongly depends on the input order of sequences. Thus, it is recommended to run the program several times with different randomized sequence input order permutations (jumbling) at high rearrangement levels for obtaining reliable results. Tree rearrangements are applied to further improve the likelihood of trees by moving all subtrees by a predefined distance (rearrangement level) within the tree. This practice has been applied to the 150 taxa data set (originally used by W. Fischer in biological research about Microsporidia) for conducting performance and scalability tests presented in [21]. However, this practice becomes prohibitive for large trees (> 200 sequences) even for parallel implementations of the above programs on supercomputers since thorough rearrangements and an augmentation in the number of taxa and jumbles leads to an increase in run time by orders of magnitude. Parallel implementations are available for the following sequential programs: **DNAml** [3], **fastDNAml** [21], **treepuzzle** [24].

As alternative to the above approaches sequential and parallel implementations of genetic algorithms for maximum likelihood-based phylogenetic tree inference have been proposed e.g. in [2] [15].

However, no comparative analysis between "traditional" and "genetic" parallel programs for maximum likelihood phylogenetic inference has been carried out so far, such that it is difficult to assess the capabilities of genetic approaches.

2. ALGORITHMIC SOLUTIONS

This section describes the algorithmic enhancements of **AxML**, which have been implemented in **fastDNAml** and **parallel fastDNAml** respectively. It describes the exhaustive algorithmic optimizations of the likelihood function of **fastDNAml** and a novel partially randomized tree building algorithm implemented in a program called **RAxML**.

2.1. Subtree column equalities (AxML)

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 induces 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:

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

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 $g(lv\_q(i), z(p,q))$.

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

For a more thorough description of SEVs see [16].

2.2. Additional SEV-based optimization (AxML2.5)

Since the initial implementation of SEVs proved to work particularly well 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 I). 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.

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

Table I. makenewz() analysis

# sequences	# makenewz() invocations	# invocations with # iterations > 1	average number of iterations if iterations > 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

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 I) 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'.

2.3. A new tree building algorithm (RAxML)

Several experimental approaches we evaluated and described in [19] for further accelerating the topology evaluation function of **AxML** did not lead to further significant run time improvements. This is due to the fact that the acceleration potential of the SEV method has been exhausted. Thus, we focus on changing the search space strategy, i.e. replacing the stepwise addition algorithm which **AxML** inherited from **fastDNAml**, by faster but equally good (in terms of attained final likelihood values) heuristics. Our new search space heuristics use a randomized approach to handle the impact of sequence input order permutations (jumbling see Section 1.1) and exploit the relationship between parsimony and maximum likelihood methods/scores [4][26]. Furthermore, we make use of consensus tree methods [7] which often improve the likelihoods of intermediate trees (see step 2) and provide valuable information for further tree refinements, which are subject of future work. Currently, our algorithm consists of three main computational steps:

1. A large number of initial trees t for a specified number $perm$ of randomized sequence input order permutations is inferred with the parsimony program **dnapars** from

PHYLIP [14], which is based on a similar stepwise addition algorithm as **fastDNAml**. The likelihood values of all final parsimony trees are calculated and the topologies are stored in an ordered tree list *otl*, of length t.

2. A majority rule consensus tree is built for all ordered fractions of *otl*, i.e. for the $2, 3, ..., t$ best trees of the list. The likelihood of each consensus tree is evaluated and inserted into *otl*.
3. Extensive local and global rearrangements are applied to the best tree of *otl*, using *exactly* the same algorithm as **fastDNAml**.

Our experiments showed, that a good setting for *perm* is twice the number of sequences n, i.e. $perm = 2n$. Note, that $t \geq perm$, since **dnapars** might yield a set of equally parsimonious trees for one single input order permutation. Among a general reduction of computation time, the algorithm yields an initial final tree containing all taxa much earlier during the inference process than **fastDNAml**.

3. TECHNICAL SOLUTIONS

This section describes various technical enhancements of the basic **PAxML** and **RAxML** codes which have been implemented to provide the means for obtaining the required amount of computational resources for the inference of large phylogenetic trees. We also focus on technical solutions which do not require expensive supercomputers and are able to exploit unused resources in workstation clusters.

3.1. Technical solutions for AxML

3.1.1. Parallel AxML

PAxML is the parallel MPI-based implementation of **AxML** which has been designed by integrating the SEV-based tree evaluation function of **AxML** into **parallel fastDNAml**. The rest of the code has remained unchanged, such that scalability and speedup are identical to **parallel fastDNAml**.

3.1.2. Distributed AxML

Distributed **AxML** is a CORBA-based implementation of **PAxML**. It uses LMC (**L**oad **M**anaged **C**ORBA [10]), which provides automatic initial object placement, migration and replication in distributed heterogeneous environments. In **DAxML** a worker object corresponds to an MPI worker process in **PAxML** and can be easily migrated or replicated since it only provides the topology evaluation function as service and does not hold a state. The distributed programming paradigm can be applied to phylogenetic tree inference, since the program passes most of its time evaluating topologies and does not communicate frequently. See [20] for a more detailed description of **DAxML**.

3.1.3. Grid AxML

Unlike traditional supercomputer applications, such as numerical simulations, **PAxML** can be quickly interrupted, checkpointed and restarted, since its state consists only of the currently best tree and information about the current phase of the computation. Thus, given the large amount of still unresolved problems in Grid computing such as e.g. the co-scheduling problem, we decided to implement a migrating Grid application. Based on the integration of the CACTUS [25] migration toolkit, **GAxML** is able to migrate through the Grid to supercomputers with free capacities. A monitoring tool for determining the best migration site is however not available yet, but this task is situated on the CACTUS-side of the application.

3.2. Technical solutions for RAxML

3.2.1. MPI-RAxML

MPI-RAxML is a straight-forward MPI implementation of the randomized algorithm described in Section 2.3.

It uses a master-worker architecture, where the master distributes the work of phases 1 through 3 and maintains the best tree list. The master controls the number of randomized input order permutations, collects respective results, and assigns fractions of the best tree list to the workers for consensus tree inference. Finally the master process generates and distributes rearranged tree topologies to the workers. In contrast to **parallel fastDNAml** however we enhanced the master with the capability to send an arbitrary number of tree topologies for evaluation to a worker instead of only one. This modification proved to be particularly useful for the design of the distributed version **RAxML@home** described in the subsequent section.

3.2.2. RAxML@home

Apart from the improvements in required run time induced by **RAxML** the algorithm has also been designed to fit the distributed programming paradigm. Thus, transformation of the **MPI-RAxML** code was particularly easy: MPI calls simply had to be replaced by the respective `http`-based routines from the library we have developed. The `http` communication library enables distributed workers to pass their messages through proxies in LANs. This library is also freely available as open source program and can be used as general purpose library for building distributed phylogenetic tree programs.

In contrast to **MPI-RAxML**, the master process of **RAxML@home** loops over a work queue which is manipulated by a dedicated interface thread.

RAxML@home already incorporates redundancy and log-file mechanisms to handle master and/or worker failures

In Figure 2 we outline a simplified program flow of **MPI-RAxML** (straight lines) and **RAxML@home** (dotted lines).

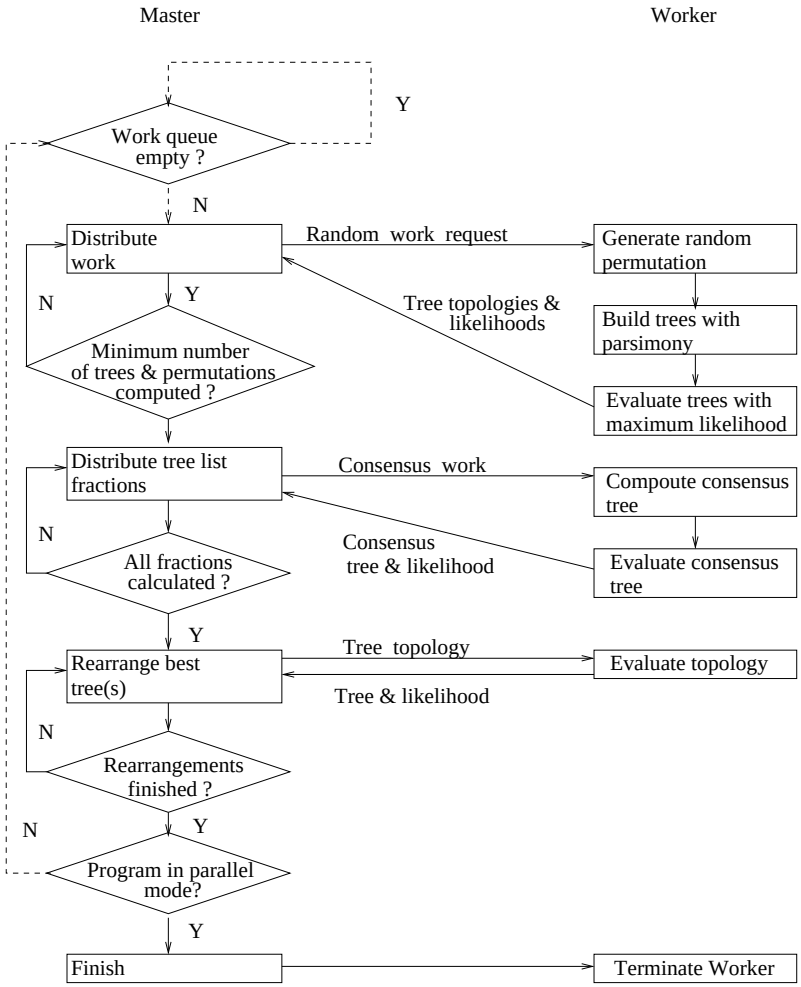


Figure 2. Outline of a simplified parallel and distributed program flow of RAxML

4. RESULTS

4.1. Test data & platforms

For testing our program we extracted alignments comprising 150, 200, 250, 500 and 1000 taxa (150_ARB,...,1000_ARB) from the ARB [11] small subunit ribosomal ribonucleic acid (ssu rRNA) database. Those alignments contain organisms from the three kingdoms Eucarya,

Table II. Total Amount of CPU hours and final ln likelihoods for PAxML and RAxML

Data	CPU hrs PAxML	ln Lh	J	R	CPU hrs RAxML	ln Lh	# trees	acc
150_SC	1635.66	-44146.90	10	5	64.29	-44145.98	500	25.44
150_ARB	300.40	-77189.78	1	5	106.14	-77189.69	500	2.83
200_ARB	774.56	-104743.33	1	5	287.57	-104743.32	500	2.69
250_ARB	1947.18	-131468.97	1	5	481.46	-131475.52	500	4.04
500_ARB	7371.79	-252588.67	1	3	2313.8766	-252617.52	500	3.19
1000_ARB	9898.05	-402282.08	1	1	1070.59	-401501.57	3837	9.23

Bacteria and Archaea. In addition, we used the 150 sequence data set (150_SC) from [21] which can be downloaded at [13].

All recent parallel tests with **RAxML** have been conducted on the HELICS [6] 512 processor Linux Cluster using 64 up to 200 processors for the largest alignments.

For testing the prototype of **RAxML@home** and **DAxML** we used various workstations at the Lehrstuhl für Rechnertechnik und Rechnerorganisation.

4.2. Algorithmic results

The run time accelerations over **(parallel) fastDNAml** achieved by **(P)AxML** on various PC processor architectures and clusters (AMD Athlon 1.4Ghz, AMD Athlon 1.6Ghz, Intel Pentium IV, Intel Xeon 2.2Ghz) exceeded 50% (max. $\approx$ 64%) for *all* alignments mentioned above and yielded *exactly* identical results [16] [17].

The run time accelerations recorded for **MPI-RAxML** over **PAxML** are listed in Table II. We list the total amount of CPU hours for each run, the final likelihood value (ln Lh) as well as the rearrangement (R) setting and the number of jumbles (J). The number of CPU hours required for the 150_SC data set is particularly large since we summed up the CPU hours of the 10 jumbled runs we conducted as specified at [13]. For **RAxML** we also list the number of randomized sequence input order trees (# trees) calculated during phase 1 of the algorithm. It is important to note that the results for the 150,...,500 sequence data sets have been attained with an older version of **RAxML** which uses standard stepwise addition without rearrangements instead of parsimony to calculate initial trees. The most important result, both in terms of run time improvement and final tree likelihood has been recorded for the 1000_ARB alignment which has been inferred with the algorithm described in Section 2.3.

The main challenge for further optimization of **RAxML** consists in the improvement of the rearrangement process since it requires approximately 80% to 90% of the total computation time.

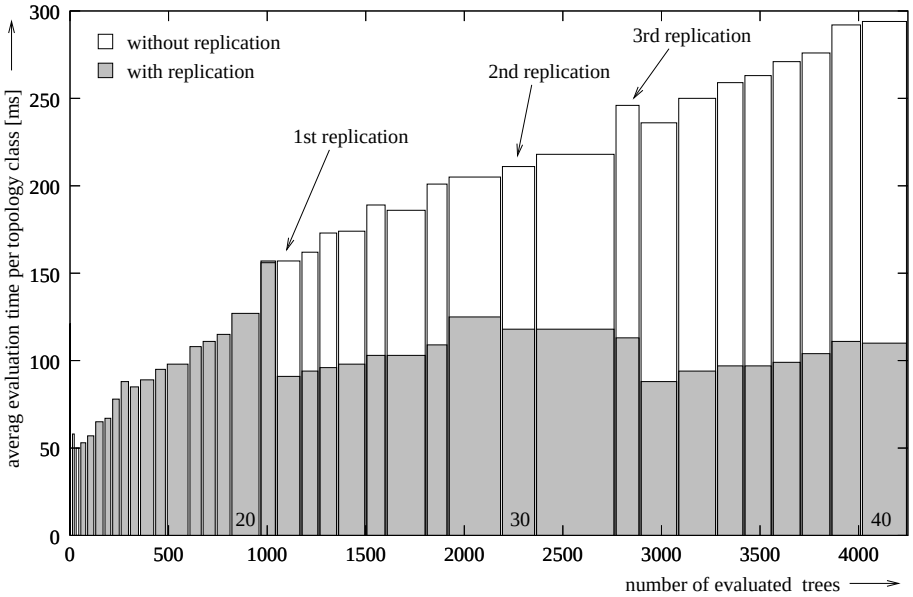


Figure 3. Impact of 3 subsequent automatic worker object replications

4.3. Some technical results

We limit the description of technical results to two examples.

The prototype of **RAxML@home** is installed and being tested on 14 processors (SOLARIS: 7 x various UltraSPARC-II and UltraSPARC-III. LINUX: 7 x various Pentium II, III, IV) at the Lehrstuhl für Rechnertechnik und Rechnerorganisation. We conducted some first successful test runs, performing however a less thorough search than for the big test runs listed in Table II. For the 150_ARB alignment we obtained a final tree with an ln likelihood of -77193.74 within approximately 12 hours. This is a promising result, since it indicates that the new distributed algorithm enables the inference of medium-sized phylogenetic trees on a comparatively small number of workstations within reasonable time.

In Figure 3 we depict the average evaluation time per topology classes 1,...,40 for two test runs with **DAxML** in the heterogeneous environment of our laboratory. A topology class is the set of all topologies of equal size, which are evaluated during a single step of the stepwise addition algorithm. We conducted one test run with automatic servant object replication enabled and another one with replication disabled.

Figure 3 shows how the average evaluation time per topology class is progressively improved by 3 subsequent automatic worker object replications (marked by arrows) performed by **LMC**.

5. Availability, current & future work

The most recent versions of **AxML** and **PAxML** are available for download at `wwwbode.in.tum.de/~stamatak/research.html`. The prototype of **RAxML@home** is developed and available at `www.sourceforge.com/projects/axml`.

Currently, we are working on the direct implementation of a parsimony score function in **RAxML** and on the integration and assessment of a parsimony-based preevaluation of topologies for the rearrangement phase.

Future work will cover improvements of the rearrangement phase based on information provided by the consensus phase of **RAxML**. This phase yields information about "strong subtrees", i.e. trees which appear in all trees of phase 1 and thus require only local optimization.

Finally, we will make an effort to port **RAxML@home** to Windows and enhance it with some fancy screen saver in order to augment the attractiveness of participation for contributors outside the academic sphere.

REFERENCES

1. Bodlaender HL, Fellows MR, Hallett MT, Wareham T, Warnow T. The hardness of perfect phylogeny, feasible register assignment and other problems on thin colored graphs. *Theor. Comp. Sci.* 2000; (244): 167–188.
2. Brauer MJ, Holder MT, Dries LA, Zwickl DJ, Lewis PO, Hillis DM. Genetic algorithms and parallel processing in maximum-likelihood phylogeny inference. *Molecular Biology and Evolution* 2002; (19): 1717–1726.
3. Ceron C, Dopazo J, Zapata EL, Carazo MJ, Trelles O. Parallel implemenation of DNAml program on message-passing architectures. *Parallel Computing* 1998; (24): 701–716.
4. DeBry RW, Abele LG. The relationship between parsimony and maximum likelihood analyses: tree scores and confidence estimates. *Mol. Biol. Evol.* 1995; (12):291–297.
5. Felsenstein J. Evolutionary trees from DNA sequences: A maximum likelihood approach. *J. Mol. Evol.* 1981; (17): 368–376.
6. HEidelberg LInux Cluster System homepage. helics.uni-hd.de [15 March 2003]
7. Jermiin LS, Olsen GJ, Mengersen KL, Easteal S. Majority-rule consensus of phylogenetic trees obtained by maximum-likelihood analysis. *Mol. Biol. Evol.* 1997; (14): 1297–1302.
8. Kannan S, Warnow T. A fast algorithm for the computation and enumeration of perfect phylogenies. *SIAM J. Comput.* 1997; **26**(6): 1749–1763.
9. Korber B et al. Timing the Ancestor of the HIV-1 Pandemic Strains. *Science* 2000; (288): 1789–1796.
10. Lindermeier M. Load Management for Distributed Object-Oriented Environments. *Proceedings of 2nd International Symposium on Distributed Objects and Applications (DOA'00)* 2000; 59–68.
11. Ludwig W et al. ARB: a software environment for sequence data. *Nucl. Acids Res.* 2003;
12. Olsen GJ, Matsuda H, Hagstrom R, Overbeek R. fastDNAml: A tool for construction of phylogenetic trees of DNA sequences using maximum likelihood. *Comput. Appl. Biosci.* 1994; (10): 41–48.
13. parallel fastDNAml donlaod site. www.indiana.edu/ rac/hpc/fastDNAml [12 February 2003].
14. PHYLIP homepage. evolution.genetics.washington.edu/phylip.html [14 April 2003].
15. Skourikhine A. Phylogenetic Tree Reconstruction Using Self-Adaptive Genetic Algorithm. In *Proceedings of IEEE International Symposium on Bio-Informatics and Biomedical Engineering (BIBE'00),* 2000.
16. Stamatakis AP, Ludwig T, Meier H, Wolf MJ. **AxML**: A Fast Program for Sequential and Parallel Phylogenetic Tree Calculations Based on the Maximum Likelihood Method. In *Proceedings of the 1st IEEE Computer Society Bioinformatics Conference (CSB 2002),* 2002;
17. Stamatakis AP, Ludwig T, Meier H, Wolf MJ. November 2002. Accelerating Parallel Maximum Likelihood-based Phylogenetic Tree Computations using Subtree Equality Vectors. In *Proceedings of Supercomputing Conference (SC2002),* 2002;
18. Stamatakis AP, Ludwig T, Meier H. Adapting **PAxML** to the Hitachi SR8000-F1 Supercomputer. In *Proceedings of 1. Joint HLRB and KONWIHR Workshop,* 2002;

19. Stamatakis AP, Ludwig T. Adapting AxML/PAxML to PC Processor Architectures. In *Proceedings of IPDPS2003, HICOMB Workshop,* 2003;
20. Stamatakis AP, Ott M, Ludwig T, Meier H. **DAxML** A program for phylogenetic tree inferenc. In *Proceedings of PaCT2003,* 2003; in press.
21. Stewart CA, Hart D, Berry DK, Olsen GJ, Wernert E, Fischer W. Parallel implementation and performance of fastDNAml - a program for maximum likelihood phylogenetic inference. In *Proceedings of of Supercomputing Conference (SC2001),* 2001;
22. Stewart CA, Tan TW, Buchhorn M, Hart D, Berry D, Zhang L, Wernert E, Sakharkar M, Fisher W, McMullen D. Evolutionary biology and computational grids. In *IBM CASCON 1999 Computational Biology Workshop: Software Tools for Computational Biology,* 1999;
23. Strimmer K, Haeseler Av. Quartet Puzzling: A Maximum-Likelihood Method for Reconstructing Tree Topologies. *Mol. Biol. Evol.* 1996; (13): 964–969.
24. Schmidt HA et al. TREE-PUZZLE: Maximum likelihood phylogenetic analysis using quartets and parallel computing. *Bioinformatics* 2002; (18): 502–504.
25. The Cactus Code. http://www.cactuscode.org [3 April 2003]
26. Tuffley C, Steel M. Links between maximum likelihood and maximum parsimony under a simple model of site substitution. *Bull Math Biol* 1997; (3):581–607.
27. Wolf MJ et al. TrExML: A maximum likelihood program for extensive tree-space exploration. *Bioinformatics* 2000; (16):383–394.