# AxML: A Fast Program for Sequential and Parallel Phylogenetic Tree Calculations Based on the Maximum Likelihood Method*

Alexandros P. Stamatakis
Technical University of Munich, Department of Computer Science
Institut für Informatik/SAB TU München, D-80290 München, Germany
stamatak@in.tum.de

Thomas Ludwig
Ruprecht-Karls-University, Department of Computer Science
Im Neuenheimer Feld 348, D-69120 Heidelberg, Germany
t.ludwig@computer.org

Harald Meier
Technical University of Munich, Department of Computer Science
Institut für Informatik/SAB, TU München, D-80290 München, Germany
meierh@in.tum.de

Marty J. Wolf
Bemidji State University, Department of Mathematics and Computer Science
Hagg-Sauer 367, 1500 Birchmont Drive, Bemidji, MN 56601-2699 USA
mjwolf@bemidjistate.edu

## Abstract

*Heuristics for the NP-complete problem of calculating the optimal phylogenetic tree for a set of aligned rRNA sequences based on the maximum likelihood method are computationally expensive. In most existing algorithms the tree evaluation and branch length optimization functions, calculating the likelihood value for each tree topology examined in the search space, account for the greatest part of overall computation time. This paper introduces* **AxML**, *a program derived from* **fastDNAml**, *incorporating a fast topology evaluation function. The algorithmic optimizations introduced, represent a general approach for accelerating this function and are applicable to both sequential and parallel phylogeny programs, irrespective of their search space strategy. Therefore, their integration into three existing phylogeny programs rendered encouraging results. Experimental results on conventional processor architectures show a global run time improvement of 35% up to 47% for the various test sets and program versions we used.*

*This work is partially 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. Introduction

At the **ParBaum** project at the Technische Universität München (TUM) work is conducted to facilitate large-scale parallel phylogenetic tree computations on trees of at least 1000 taxa on the Hitachi SR8000-F1 supercomputer installed at the Leibniz-Rechenzentrum (LRZ) in Munich. Our work relies on sequence data provided by the ARB [9] rRNA-sequence database, developed jointly by the Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR) and the Department of Microbiology of the TUM. The ARB database provides a huge amount of sequence data with excellent alignment quality.

Like many problems associated with genome analysis, the perfect phylogeny problem is NP-complete. 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 ($\approx$ 500 taxa [7] [8]), compared to the huge amount of data available ($\approx$ 20000 sequences in the ARB database), have been calculated so far.

We focus on *three* key areas to attain our goal of producing large, high quality evolutionary trees:

1. *Improvement of the existing algorithms* by introduction of new heuristics and algorithmic optimizations.

2. Adaptation of the existing algorithms to *hybrid supercomputer architectures*.

3. Integration of *empirical biological knowledge* into algorithms.

This paper mainly presents results concerning algorithmic optimizations for accelerating the computation of the topology evaluation function and describes an initial adaptation of the parallel program to the Hitachi SR8000-F1 supercomputer, i.e. covers points 1 and 2.

The optimizations are applicable to most existing sequential and parallel programs, for phylogenetic tree inference based on the maximum likelihood method, especially derivatives of **fastDNAml** [6] [11] and the **phylip** [12] [4] package and are independent from the specific search space strategy.

We implemented the optimizations proposed in this paper in **AxML** (A(x)cce-lerated Maximum Likelihood) and **PAxML** (Parallel AxML) based on the latest sequential and parallel releases of **fastDNAml** (v.1.2.2).

Our initial experiments on conventional processor architectures obtained total run time reductions ranging from 35% to 47%, both for the sequential, as well as the parallel program. Furthermore, **PAxML** has already been appropriatly adapted to the Hitachi SR8000-F1 supercomputer architecture and rendered 30% to 35% performance improvment for sufficiently large data sets.

These results are promising first steps toward efficient determination of large, high quality evolutionary trees using supercomputers. In addition, we have demonstrated the generality of our approach by incorporating our optimization into **TrExML**, a program with a more extensive tree space exploration strategy than **fastDNAml**. We call the resulting program **A**ccelerated **TrExML** (**ATrExML**). Initial experiments with **ATrExML** have shown analogous performance improvements over **TrExML** to those mentioned above.

## 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, over look 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.

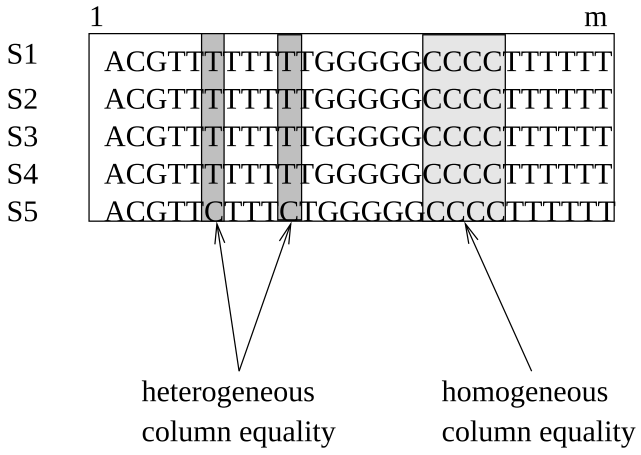
1
m
S1 ACGTTTTTTTTGGGGGCCCCTTTTTT
S2 ACGTTTTTTTTGGGGGCCCCTTTTTT
S3 ACGTTTTTTTTGGGGGCCCCTTTTTT
S4 ACGTTTTTTTTGGGGGCCCCTTTTTT
S5 ACGTTCTTTCTGGGGGCCCCTTTTTT
heterogeneous column equality
homogeneous column equality

**Figure 1. Heterogeneous and homogeneous columns**

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 (see figure 1).

More formally, let $s_1, ..., s_n$ be the set of aligned input sequences as depicted in the upper matrix of figure 2.

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, as depicted in figure 2.

Since a necessary prerequisite for a phylogenetic tree calculation is a high-quality multiple alignment of the input

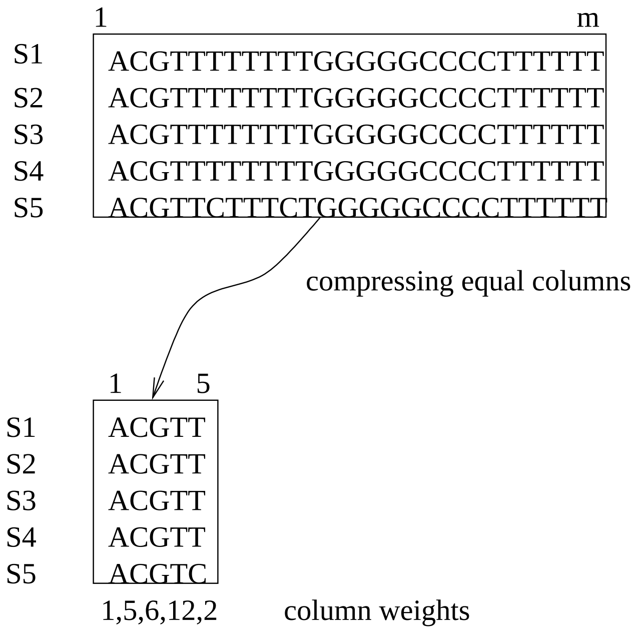
1
m
S1
ACGTTTTTTTTGGGGGCCCCTTTTTT
S2
ACGTTTTTTTTGGGGGCCCCTTTTTT
S3
ACGTTTTTTTTGGGGGCCCCTTTTTT
S4
ACGTTTTTTTTGGGGGCCCCTTTTTT
S5
ACGTTCTTTCTGGGGGCCCCTTTTTT
compressing equal columns
1
5
S1
ACGTT
S2
ACGTT
S3
ACGTT
S4
ACGTT
S5
ACGTC
1,5,6,12,2
column weights

**Figure 2. Global compression of equal columns, all column weights are 1 in the uncompressed matrix**

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 (see example in figure 3):

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

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,

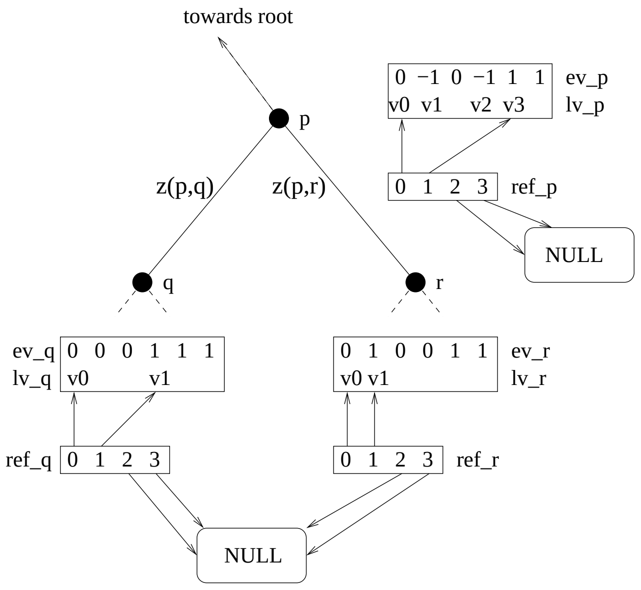

**Figure 3. Example likelihood-, equality- and reference-vector computation for the subtree at p**

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 a further significant reduction in the number of floating point operations by extending this mechanism to the branch length optimization function.

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

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

A simple example for the optimized likelihood vector calculation and the respective data-types used is given in figure 3.

## 3. Implementation

We integrated subtree equality vectors into three existing phylogeny programs: **fastDNAml** [6], **fastDNAmlP** [7] and **TrExML** [13]. We name the optimized versions **AxML**, **PAxML** and **ATrExML** respectively. About 300 lines of code have been added to the various programs, thus demonstrating the efficiency, simplicity and applicability of our approach.

A simple analysis of **fastDNAml** with the `gprof` tool shows that the tree likelihood function `newview()` and the branch length optimization function `makenewz()` consume over 95% of overall execution time. The basic ideas of this paper have been implemented in functions `newview()`, `makenewz()`, `sigma()` and `evaluate()`, since those functions access the likelihood-vectors of the nodes and are affected by the changes induced by skipping assignments of type $lv\_p(i) = lv\_p(j), i < j, ev\_p(j) = ev\_p(i) > -1$.

In each of those functions the main for-loop over the sequence length $m'$ has been modified in order to correspond to formula 3 and the code for calculating the equality vector values has been added. Furthermore, an additional loop for initializing $precalc\_q(c)$ and $precalc\_r(c)$ has been inserted.

The remaining modifications concern initialization matters, and the definition of a few additional data-types for storing the $precalc()$ and $ref()$ array information.

## 4. Adaptation to the Hitachi SR8000-F1

For initial testing on the Hitachi SR8000-F1 we chose to use inter-node MPI, i.e. all 8 processors forming part of one shared memory node, are used independently and are assigned one MPI process each.

The first tests in this configuration rendered less impressive results in terms of run time improvement of **PAxML** over **fastDNAml**, compared with the results obtained on conventional processor architectures (see section 5). The problem could however be quickly identified. The case analysis of formula 3 has originally been implemented within the computationally expensive for-loops of functions `newview()`, `makenewz()`, `sigma()` and `evaluate()` as nested conditional statement. This implementation scales well on conventional architectures but in contrast significantly perturbs the pipelining and prefetch mechanisms of Hitachi's hardware architecture.

Therefore, we split up the for-loops within the functions mentioned above, and implemented a distinct for-loop for each case, thus avoiding further conditional statements within the respective loops. We inserted a precalculation step where the equality vector values are computed and calculate at the same time a reference array for each distinct case in 3. In addition to this, the number of entries for each case is counted, such that a distinct for-loop for each case can be constructed, which accesses the likelihood vector via its respective reference array.

This modification boosted program efficiency, both, in terms of floating point performance and run time reduction, although some additional code had to be inserted for precalculating the loop split and accessing the likelihood vector through reference arrays.

E.g. the non-adapted **PAxML** code rendered 25.47% run time improvement compared with 34.71% for the adapted one with the 41 taxa mitochondrial rRNA test set executed on two workers (see section 5).

## 5. Results

The amount of performance improvement strongly depends on the number and length of the input sequences, as well as the quality of the alignment. We note that whenever more subtree column equalities are expected, performance improves more. We establish two general rules:

1. Performance improves with the quality of the alignment.
2. Performance improves with the length of the sequences.

We initially present the results and tests performed on conventional architectures and report about first results on the Hitachi SR8000-F1 at the end of this section.

We tested the performance of **AxML**, **PAxML** and **ATrExML** with data sets from various sources and obtained global run time improvements between 35% and 47%. We compiled the sequential programs with `gcc -O3` and executed them under `Solaris` on a `Sun-Blade-1000`. For the parallel programs we used `gcc -O2` with the master and foreman components located on a `Sun-Blade-1000` and two workers, each running on a `Sun Ultra 5/10`.

For analyzing the global run time improvement of **AxML/PAxML** over **fastDNAml/fastDNAmlP** tests with data-sets of 20, 30, 40 and 50 taxa (sequence length: 840) extracted from the alignment of 56 sequences delivered as the test-set with `fastDNAmlP`, as well as two alignments consisting of 161 mitochondrial rRNA sequences (sequence length: 511) from beetles and butterflies were used. We used different program options and data-sizes for demonstrating the scalability and generality of our method. In table 1 we present the global run time improvement of **AxML/PAxML** over **fastDNAml/fastDNAmlP**, both with the quickadd (local branch length optimization) option enabled and disabled. For the mitochondrial rRNA test sets we executed tests with and without tree rearrangements (for details refer to the **fastDNAml** documentation).

Results for the 20 to 50 taxa sequential test runs are also depicted in figure 4. An important result is that **AxML** with the quickadd option disabled, still runs about 15% to 20% faster than **fastDNAml** with the quickadd option enabled, thus ensuring a higher tree quality. Furthermore, the algorithmic optimizations scale well to **PAxML**, even in the case of considerably small test sets for a parallel run as in the case of the 20 to 50 taxa test set. The results obtained from the 161 taxa tests demonstrate the scalability of our

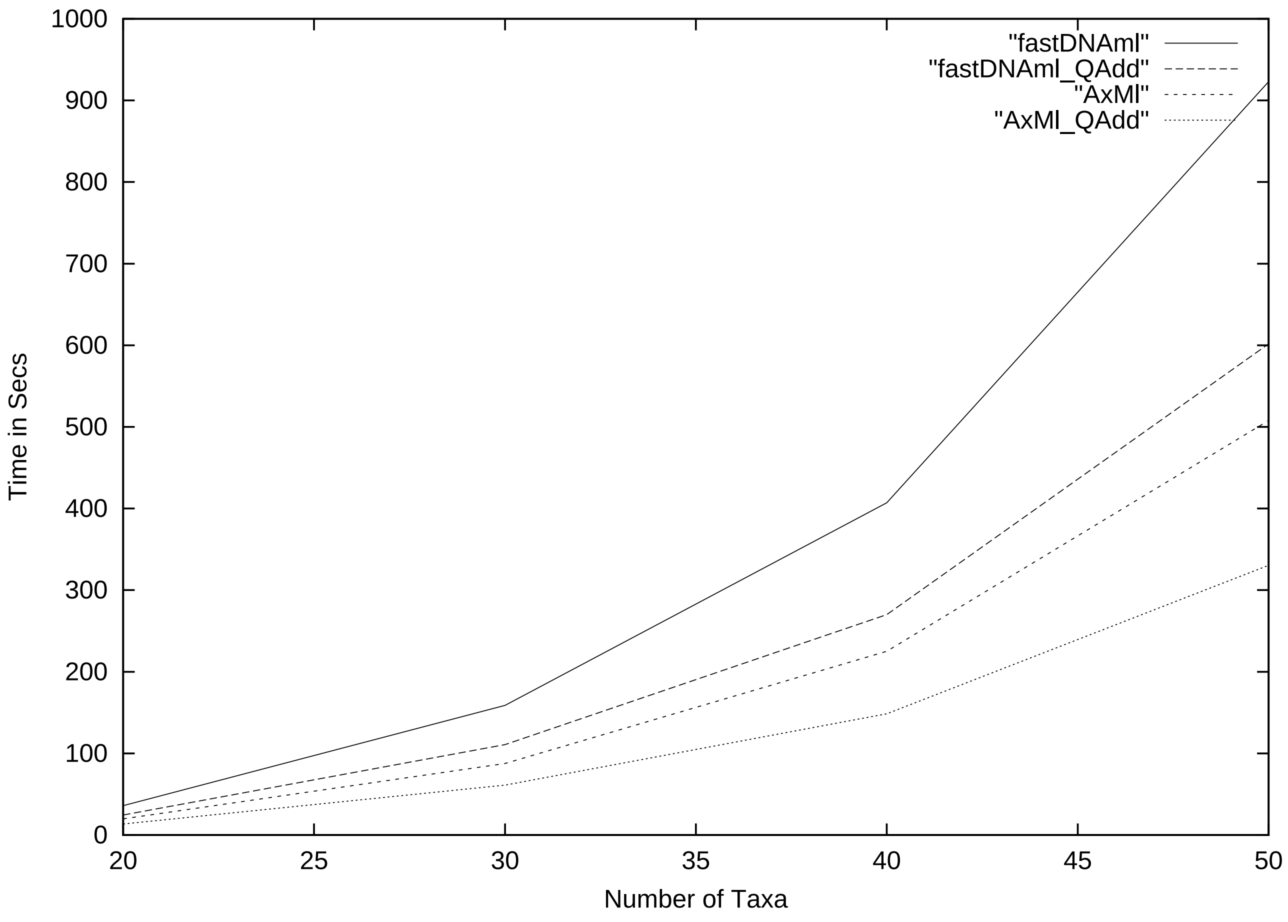

**Figure 4. Overall execution times of fastDNAml and AxML with the quickadd option enabled and disabled**

approach, both to large test sets, i.e. with a great number of taxa, as well as to the parallel algorithm. The good parallel performance improvement is due to the fact that the tree evaluation function is the core of the worker components, which perform the actual computation (for details refer to [7]).

For the performance analysis of **ATrExML** versus **TrExML**, presented in table 2 we used the same data-sets as in the original **TrExML** publication. For details on the parameters and data used refer to [13].

Since **AxML** does not implement heuristics but only a purely algorithmic optimization in all tests **AxML** and **fastDNAml** rendered exactly the same results, a fact that can be verified by a simple `diff` on the output files.

For the initial tests performed on the SR8000-F1 we used the same 30, 40, 50 taxa input data as in the previous tests and an additional 56 taxa test set. Furthermore we also used a 41 taxa alignment (sequence length: 1500) of mitochondrial rRNA sequences from beetles and butterflies, similar

| data set | option | **AxML** | **PAxML** |
|---|---|---|---|
| 20 taxa | Qadd | 44.91% | 36.74% |
| 30 taxa | Qadd | 44.81% | 37.93% |
| 40 taxa | Qadd | 44.91% | 37.30% |
| 50 taxa | Qadd | 45.09% | 38.01% |
| 20 taxa | No Qadd | 44.95% | 35.17% |
| 30 taxa | No Qadd | 44.77% | 38.06% |
| 40 taxa | No Qadd | 44.72% | 37.58% |
| 50 taxa | No Qadd | 44.97% | 36.96% |
| 161 taxa.1 | rearr. | 46.90% | 39.32% |
| 161 taxa.2 | rearr. | 46.98% | 39.24% |
| 161 taxa.1 | No rearr. | 39.48% | 37.81% |
| 161 taxa.2 | No rearr. | 39.53% | 39.03% |

**Table 1. Global run time improvements fastDNAml(p) vs. (P)AxML**

| a | n | improvement | a | n | improvement |
|---|---|---|---|---|---|
| 8 | 10 | 38.21% | 8 | 11 | 38.67% |
| 8 | 12 | 39.58% | 8 | 13 | 40.02% |
| 8 | 14 | 39.87% | 8 | 15 | 40.68% |
| 8 | 16 | 40.71% | 9 | 10 | 38.24% |
| 9 | 11 | 38.91% | 9 | 12 | 39.91% |
| 9 | 13 | 40.77% | 9 | 14 | 40.19% |
| 9 | 15 | 41.04% | 9 | 16 | 41.10% |
| 10 | 10 | 37.23% | 10 | 11 | 38.39% |
| 10 | 12 | 39.94% | 10 | 13 | 41.08% |
| 10 | 14 | 42.26% | 10 | 15 | 43.00% |
| 10 | 16 | 42.26% | | | |

**Table 2. Global run time improvements TrExML vs. ATrExML**

to the 161 taxa test set mentioned above. The latter was used for analyzing the scalability of our optimization concerning long sequences. We compiled both, **fastDNAmlP** and **PAxML** with `mpicc -O3 -model=F1`, measured Mflops/s/processor(worker) performance using `PCL`, and executed the test runs with 2 up to 4 workers located on the same node in intra-node MPI-mode. The results including the respective program options are presented in table 3. We performed those tests with the rearrangement and quickadd options enabled. Those results are encouraging, since the algorithmic optimizations of **PAxML** scale well to the specific hardware architecture, especially when longer sequences are used, as will be the case for production runs using data from the ARB. Furthermore, the Mflops/s/processor(worker) rate is already satisfying (e.g. 50 Mflops/s/processor are considered to be a "good" value for an initial test run by the Hitachi SR8000-F1 administration), although no special compiler options for further optimizing the code have been used so far. The Mflops/s/processor performance varies less than 0.5% among the different workers, i.e. there is no load balancing problem.

| data set | workers | **PAxML** | Mflops/s/proc. |
|---|---|---|---|
| 30 taxa | 2 | 24.82% | 123.77 |
| 40 taxa | 2 | 28.10% | 128.06 |
| 50 taxa | 2 | 27.76% | 128.62 |
| 56 taxa | 2 | 27.10% | 128.55 |
| 41 taxa | 2 | 34.17% | 119.74 |
| 41 taxa | 4 | 30.17% | 106.51 |

**Table 3. Global run time improvement PAxML vs. fastDNAmlP and Mflops/s/proc. performance on the Hitachi SR8000-F1**

## 6. Availability

The most recent distribution versions of **AxML**, **PAxML** and **ATrExML** are available for download at [10]. We also provide a program called Simple **PAxML**, that may be used for developing CORBA and PAxML@home-like (see sections 7 and 8) applications and provides a more adequate program structure for developing such kind of programs. A **PAxML** distribution version including a compiler switch for using the loop transformations for supercomputers will soon be released.

## 7. Current Work

Currently we focus on the evaluation of different Hitachi-specific parallelization concepts, such as pseudo-vectorization or inter-node MPI and the respective adaptation of the program, as well as on the preparation and execution of first production test runs using sequence data from the ARB database.

For estimating the expected run time improvement induced by **AxML** for a specific input data set we have already developed an efficient best-case/worst-case estimate algorithm, which is presently being implemented. Note, that a quick estimate for a specific sequence alignment might also be obtained by executing **AxML** and **fastDNAml** without local and global rearrangements, since the programs execute significantly faster in that configuration (see section 8).

Furthermore, we are continuously extending the **AxML** program family and investigate the applicability of various programming paradigms for handling the complexity of the problem beyond the scope of traditional supercomputing. Within this context we are currently developing **DAxML** (Distributed **AxML**) and **GAxML** (Grid **AxML**).

**DAxML** is a CORBA (Common Object Request Broker) version of **PAxML** based on **LMC** (Load Managed CORBA [5]). **LMC** is an automatic load balancing tool for CORBA applications which is integrated transparently into the ORB (Object Request Broker) and distributes load by initial servant object placement, migration and replication. The initial version of **DAxML** has already been successfully tested on the Sun-cluster of the LRR. It performs well, both in terms of performance, i.e. CORBA overhead and load distribution, since the parallel algorithm of **PAxML** is well suited for distributed computation. Presently we are developing a standard CORBA distribution version of **DAxML**, i.e. without load balancing, based on a freely available ORB implementation.

**GAxML** is a "phylogenetic grid worm", which is being developed in cooperation with the Cactus team [2][1] at the Max-Planck-Institut f ̈ur Gravitationsphysik, Albert-Einstein-Institut.

Unlike many typical supercomputer applications, interrupting, checkpointing and restarting **GAxML** is very fast, and the state to be saved comprises only a few lines of ASCII text. Since the co-scheduling problem has not yet been resolved in practice and communication between supercomputers at different sites might have a negative impact on performance [7][8], we invented the term "phylogenetic grid worm" for an application migrating to sites with free capacities on the grid, for avoiding those problems.

**GAxML** is also based on **PAxML** and already incorporates the necessary modifications for initiating a migration request and a compiler switch for selecting the appropriate case-switch implementation (see section 4) for a specific architecture, e.g. a classical supercomputer or a huge Linux cluster. Migrations will be performed by using a grid migration server developed by the Cactus team.

## 8. Future Work

Future work will cover the implementation and analysis of a different parallelization approach.

An important fact within this context is, that **AxML** runs significantly faster with the local and global rearrangement option switched off (E.g. factor 100 for the 161 mitochondrial rRNA sequences). Simply switching off rearrangements may decrease tree quality on the one hand, but facilitates the analysis of a greater number of input sequence permutations on the other hand, which in turn increases tree quality again.

Since the calculation of a single tree without rearrangements becomes significantly faster one can distribute input sequence permutations instead of tree topologies to the workers, thus significantly reducing the communication and synchronization overhead. Furthermore, the similarity among the sequence of generated topologies during tree reconstruction, is greater without performing rearrangements and therefore, there is a potential for additional algorithmic optimizations.

We plan to implement a two-step parallel algorithm, which initially performs tree reconstruction as described above and then executes the standard **PAxML** algorithm with rearrangements in the second step, using those sequence permutations, that rendered the best trees in step one. Within this context we will evaluate various strategies and algorithms for generating useful input sequence permutations.

Finally, we plan to develop and distribute PAxML@home, an application in the style of the very successful project SETI@home [3], based on peer-to-peer communication, the fast algorithm of **PAxML** and the experience gained with the development of **DAxML**.

## References

[1] G. Allen, W. Benger, T. Dramlitsch, T. Goodale, H.-C. Hege, G. Lanfermann, A. Merzky, T. Radke, and E. Seidel. Cactus grid computing: Review of current development. *LNCS*, 2150:817 ff., 2001.

[2] G. Allen, W. Benger, T. Dramlitsch, T. Goodale, H.-C. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shal. Cactus tools for grid applications. *Cluster Computing*, 4(3):179–188, 2001.

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

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

[5] M. Lindermeier. Load management for distributed object-oriented environments. In *Proceedings of 2nd International Symposium on Distributed Objects and Applications (DOA'00)*, pages 59–68, 2000.

[6] 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.

[7] 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.

[8] C. Stewart, T. Tan, M. Buchhorn, D. Hart, D. Berry, Z. L., E. Wernert, M. Sakharkar, W. Fisher, and D. McMullen. Evolutionary biology and computational grids. Technical report, IBM CASCON Computational Biology Workshop: Software Tools for Computational Biology, 1999.

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

[10] TUM. Axml, paxml, atrexml download site. Technical report, WWWBODE.IN.TUM.DE/˜STAMATAK/RESEARCH.HTML, 2002.

[11] UIUC. fastdnaml distribution. Technical report, GETA.LIFE.UIUC.EDU/˜GARY/PROGRAMS, 2002.

[12] UW. The phylogeny inference package. Technical report, EVOLUTION.GENETICS.WASHINGTON.EDU/PHYLIP.HTML, 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.