Accelerating Parallel Maximum Likelihood-based Phylogenetic Tree Calculations using Subtree Equality Vectors *

Alexandros P. Stamatakis
Technical University of Munich, Department of Computer Science
stamatak@in.tum.de

Thomas Ludwig
Ruprecht-Karls-University, Department of Computer Science
thomas.ludwig@informatik.uni-heidelberg.de

Harald Meier
Technical University of Munich, Department of Computer Science
meierh@in.tum.de

Marty J. Wolf
Bemidji State University, Department of Mathematics and Computer Science
mjwolf@bemidjistate.edu

Abstract

Heuristics for calculating a phylogenetic tree for a large set of aligned rRNA sequences based on the maximum likelihood method are computationally expensive. The core of most parallel algorithms, which accounts for the greatest part of computation time, is the tree evaluation function, calculating the likelihood value for each tree topology. This paper introduces a general method to accelerate this function by using Subtree Equality Vectors (SEVs) for reducing the number of required floating point operations during topology evaluation.

We integrated our optimizations into various sequential programs and into parallel fastDNAml, one of the most common and efficient parallel programs for calculating large phylogenetic trees.

Experimental results for our parallel program, which renders exactly the same output as fastDNAml show global run time improvements of 26% up to 65%. The optimization scales best on clusters of PCs, which also implies a substantial cost saving factor for the determination of large trees.

© 0-7695-1524-X/02 $17.00 (c) 2002 IEEE

*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”: Kompetenzzentrum für Technisch-Wissenschaftliches Hoch- und Höchstleistungsberechnen in Bayern (KONWIHR). KONWIHR is funded by means of “High-Tech-Offensive Bayern”.
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 [8] 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. 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.

At the ParBaum (Parallel Tree) 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 [6] (rank 14 in the top 500 supercomputers list, June 2002 [15]) installed at the Leibniz-Rechenzentrum (LRZ) in Munich. Our work relies on alignments from the small subunit ribosomal Ribonucleic Acid (ssrRNA) database of the ARB project [13], developed jointly by the Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR) and the Department of Microbiology at the TUM. The ARB database provides high quality alignments of 16S and 18S rRNA sequences.

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 [7]. Like many problems associated with genome analysis, 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 (≈ 500 taxa [11, 12]), have been calculated so far, although large data sets containing potential phylogenetic information are available (> 20000 sequences in the ARB ssrRNA database).

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

1. Integration of empirical biological knowledge into algorithms.

2. Adaptation of the existing algorithms to hybrid supercomputer architectures.
3. Improvement of the existing algorithms by introduction of new heuristics and algorithmic optimizations.

This paper presents results concerning algorithmic optimizations for accelerating the computation of the topology evaluation function used by maximum likelihood-based programs. These optimizations are applicable to most existing sequential and parallel programs for phylogenetic tree inference based on the maximum likelihood method, especially derivatives of *fastDNAml* [3, 9] and the *phylip* [4, 14] package, and are independent from the specific search space strategy.

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

Apart from the description of the algorithmic optimization, the performance of the parallel version for large test sets and the impact of different hardware architectures on its efficiency is our main focus.

Our experiments with *Paxml* obtained total run time reductions ranging from 26% to 65% and demonstrate that our approach scales well to the parallel program.

The good scalability to the parallel program is due to the fact that most parallelization approaches [2, 11] are based on a master-worker architecture, with the workers performing the topology evaluation tasks, i.e., the tree evaluation function is the core of the worker process.

The big deviation in obtained run time improvement is mainly due to different hardware architectures, and our results show that the optimization scales especially well on cheap processor architectures, equipped with less powerful FPGAs.

These results are promising first steps toward efficient determination of large, high quality evolutionary trees using supercomputers and big, less expensive clusters of PCs, since we have significantly accelerated a program that has already been used for large scale phylogenetic tree computations on supercomputers [11]. Furthermore, we have access to one of the most powerful supercomputers worldwide for conducting production runs of biological importance and will soon obtain access to the HEidelberg Linux Cluster System (HELICS [6]), installed at the Institut für Wissenschaftliches Rechnen (IWR) Heidelberg, which is presently the largest Linux cluster worldwide (rank 35 in the top 500 supercomputers list, June 2002 [15]).

In addition, we provide our programs free of charge to the community via the WWW.

Finally, 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 Accelerated *TrExML* (*ATrExML*). Initial experiments with *ATrExML* have shown performance improvements over *TrExML* analogous to those mentioned above [10].

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.

![Diagram of column compression](image)

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

More formally, let \( s_1, \ldots, s_n \) be the set of aligned input sequences as depicted in the upper matrix of Figure 1.

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 \( \forall s_k, k = 1, \ldots, n : s_{ki} = s_{kj} \), 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 1.

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 $e^k$ different column equality classes, where $k$ is the number of tips (sequences) in the subtree of $p$ and $e$ 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 $e > m'$ where $m'$ is the length of the compressed global sequences.

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 $m'$. 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 2):

$$
\text{ev}_p(i) := \left\{ \begin{array}{ll}
\text{ev}_q(i) & \text{if } \text{ev}_q(i) = \text{ev}_r(i) \\
-1 & \text{else}
\end{array} \right.
$$

(1)

If $p$ is a leaf, we set $ev_p(i) := map(\text{sequence}_p(i))$, where, $map()$ is a function that maps the character representation of the aligned input sequence $\text{sequence}_p$, at leaf $p$ to values $0, 1, ..., e$. Thus, the values of an inner SEV $ev_p$, at position $i$, range from $-1, 0, ..., e$, i.e. $-1$ if column $i$ is heterogeneous and from $0, ..., e$ in the case of an homogeneous column.

For SEV values $0, ..., e$ a pointer array $\text{ref}_p(e)$ 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 $\text{ref}_p(ev_p(j)) \neq NULL$ for an index $j$ of the likelihood vector $\text{lv}_p(j)$ 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 $\text{lv}_p(j) := \text{lv}_p(i)$. If $ev_p(j) > -1$ and $\text{ref}_p(ev_p(j)) = NULL$, we assign $\text{ref}_p(ev_p(j))$ to the address of $\text{lv}_p(j)$, i.e. $\text{ref}_p(ev_p(j)) := add(\text{lv}_p(j))$. 

5
The additional memory required for equality vectors is $O(n \times m')$. 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%\(^1\). 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.

\(^1\)The percentages mentioned in this section were obtained during initial tests and program development on a Sun-Blade-1000.
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, since 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 \( p, lv_{\varphi}(i) \), is calculated:

\[
lv_{\varphi}(i) := f(g(lv_{\varphi}(i), z(p, q)), g(lv_{\varphi}(i), z(p, r)),
\]

where \( z(p, q) \) (\( z(p, r) \)) is the length of the branch from \( p \) to \( q \) (\( p \) to \( r \), respectively).
The function \( g() \) is a computationally expensive function, that calculates the likelihood
of the left (right) branch of \( p \), depending on the branch length \( z(p, q) \) (\( z(p, r) \)) and the
value of \( lv_{\varphi}(i) \) (\( lv_{\varphi}(i) \), respectively). Whereas \( f() \) performs some simple arithmetic
operations for combining the results of \( g(lv_{\varphi}(i), z(p, q)) \) and \( g(lv_{\varphi}(i), z(p, r)) \)
into the value of \( lv_{\varphi}(i) \). Note that \( z(p, q) \) and \( z(p, r) \) do not change with \( i \).

If we have \( ev_{\varphi}(i) > -1 \) and \( ev_{\varphi}(i) = ev_{\varphi}(j), i < j \), we have \( lv_{\varphi}(i) = lv_{\varphi}(j)
and therefore \( g(lv_{\varphi}(i), z(p, q)) = g(lv_{\varphi}(j), z(p, q)) \) (the same equality holds for node
\( r \)). Thus, for any node \( q \) we can avoid the recalculation of \( g(lv_{\varphi}(i), z(p, q)) \) for all
\( j > i \), where \( ev_{\varphi}(j) = ev_{\varphi}(i) > -1 \). We precalculate these values and store them
in arrays \( precalc_{\varphi}(c) \) and \( precalc_{\varphi}(c) \) respectively, where \( c \) is the number of distinct
color-value mappings found in the sequence alignment.

Our final optimization consists in the elimination of value assignments of type
\( lv_{\varphi}(i) := lv_{\varphi}(j) \), for \( ev_{\varphi}(i) = ev_{\varphi}(j) > -1, i < j \) where \( i \) is the first entry for a specific
homogeneous equality class \( ev_{\varphi}(i) = 0, ..., c \) in \( ev_{\varphi} \). We need not assign these values
due to the fact that \( lv_{\varphi}(j) \) will never be accessed. Instead, since \( ev_{\varphi}(j) = ev_{\varphi}(i) > -1 \)
and the value of \( g_{\varphi}(j) = g_{\varphi}(i) \) has been precalculated and stored in \( precalc_{\varphi}(ev_{\varphi}(i)) \),
we access \( lv_{\varphi}(i) \) through its reference in \( ref_{\varphi}(ev_{\varphi}(i)) \).

During the main for-loop in the calculation of \( lv_{\varphi} \) we have to consider 6 cases,
depending on the values of \( ev_{\varphi} \) and \( ev_{\varphi} \). For simplicity we will write \( p_{\varphi}(i) \) instead of
\( precalc_{\varphi}(i) \) and \( g_{\varphi}(i) \) instead of \( g(lv_{\varphi}(i), z(p, q)) \):

\[
\begin{align*}
    lv_{\varphi}(i) := & \quad \begin{cases}
    f(p_{\varphi}(ev_{\varphi}(i)), p_{\varphi}(ev_{\varphi}(i))) & \text{if } ev_{\varphi}(i) = ev_{\varphi}(i) > -1, \quad ref_{\varphi}(ev_{\varphi}(i)) = \text{NULL} \\
    \text{skip} & \text{if } ev_{\varphi}(i) = ev_{\varphi}(i) > -1, \quad ref_{\varphi}(ev_{\varphi}(i)) \neq \text{NULL} \\
    f(p_{\varphi}(ev_{\varphi}(i)), p_{\varphi}(ev_{\varphi}(i))) & \text{if } ev_{\varphi}(i) \neq ev_{\varphi}(i), \quad ev_{\varphi}(i), ev_{\varphi}(i) > -1 \\
    f(p_{\varphi}(ev_{\varphi}(i)), g_{\varphi}(i)) & \text{if } ev_{\varphi}(i) > -1, ev_{\varphi}(i) = -1 \\
    f(g_{\varphi}(i), p_{\varphi}(ev_{\varphi}(i))) & \text{if } ev_{\varphi}(i) > -1, ev_{\varphi}(i) = -1 \\
    f(g_{\varphi}(i), g_{\varphi}(i)) & \text{if } ev_{\varphi}(i) = -1, ev_{\varphi}(i) = -1
    \end{cases}
\end{align*}
\]

A simple example for the optimized likelihood vector calculation and the respective
data-types used is given in Figure 2.
2.1 Parallelization Aspects

Most parallel implementations of maximum likelihood-based phylogeny programs use a coarse-grained parallelization approach by distributing a set of topologies among the workers for evaluation. A fine-grained parallelization approach, i.e., the parallelization of the tree evaluation (likelihood) function itself, suffers from various disadvantages.

A typical approach for a shared memory architecture would consist of splitting up the likelihood vectors of one tree into equal segments among the processors. A great part of partial likelihood vector computation can be performed independently during recursive tree traversal but synchronization is required at those points of the program where the entire likelihood vector has to be available, for calculating the likelihood value, e.g., at the branches where branch length optimization is being performed.

In the special case of our optimization the workload among likelihood vector segments may vary significantly, depending on the distribution of equality vector entries in the respective SEV segments, such that an additional performance penalty has to be expected, due to load imbalance. This could be resolved by dynamically calculating an appropriate asymmetric likelihood vector split at each node of the tree, which however induces an additional overhead.

In addition, the completion of the precalculation step for obtaining $precalc(v(c))$ becomes a precondition for the execution of the main for-loop of the critical functions, when using SEVs.

Therefore, we decided to maintain the coarse-grained parallelization scheme, even on the hybrid architecture of the Hitachi supercomputer, since it ensures optimal efficiency and scalability of our algorithmic optimization.

3 Implementation

We integrated subtree equality vectors into three existing phylogeny programs: fastDNAml [9], parallel fastDNAml [11] and TrExML [16]. We name the optimized versions AXML, PAXML and AtExML respectively. About 300 lines of code (≈ 5% in the sequential code) have been added to those 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 makenew() consume over 95% of overall execution time. The basic ideas of this paper have been implemented in functions newview(), makenew(), 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 $ev(i) := ev(j), i < j, ev(j) = ev(i) > -1$.

In each of those functions the main for-loop over the number of distinct columns $m'$ has been modified in order to correspond to formula 3. Furthermore, an additional loop of constant length $\leq c$ for initializing $precalc(v(c))$ and $precalc(v(c))$ has been inserted.

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

For PAXML we designed a special version consisting of a single binary (paxml) instead of three distinct ones (master, foreman, slave), for reasons of portability, since the execution of multiple binaries is not supported by all MPI environments.
3.1 Adaptation to the Hitachi SR8000-F1

Testing PAxML on the Hitachi SR8000-F1 showed that its execution is less efficient than on more conventional architectures (see section 4) and that the amount of performance improvement strongly depends on the specific processor architecture.

The observed behavior is clearly associated with the hardware architecture of the SR8000-F1 and with the case analysis of formula 3 in particular, which has originally been directly inserted into the computationally expensive for-loops of functions newview(), makeswzz(), sigsa() and evaluate() as nested conditional statement.

A modification of the program, where we split up the main for-loop of the above functions into distinct (smaller in length) for-loops for each case of formula 3, for avoiding conditional statements within loops confirms this hypothesis.

The modified version performs slightly better (e.g. 6% faster for the 150 taxa test set, see section 4) than the initial one, although a significant overhead is induced by the loop split due to some additional precalculations and indexing operations.

Furthermore, the SR8000-F1 is the only architecture where the above modification rendered better results as the original version.

On all other processor architectures we used for testing AXML and PAxML (Intel Pentium II & III, Sun UltraSPARC-III, SGI/Cray Origin 2000, AMD Athlon MP) the version with loop splits performs significantly worse.

4 Results

The amount of expected run time improvement of AXML and PAxML in relation to fastDNAml and parallel fastDNAml respectively depends on two main factors.

Firstly, 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.

Secondly, our latest tests reveal a strong correlation between processor architecture and program performance. We observe that the run time reduction percentage can vary between 26% and 65% for the same test set on different architectures.

In this section we focus mainly on the performance of PAxML for larger sequence alignments, the impact of processor architecture on performance and an ad hoc solution for predicting the expected performance improvement.

For results concerning AXML, ATExML, initial small tests with PAxML as well as the impact of various program options on program performance refer to [10].

4.1 Platforms and Compilers

We used two different platforms for conducting large parallel test runs:
1. A Myrinet-based 8 node Linux cluster equipped with 16 Intel Pentium III processors and 2 Gbyte of main memory. The processors we used were exclusively reserved for our experiments.

2. The Hitachi SR-8000F1, where we exclusively reserved 2 nodes, i.e. 16 processors in intra-MPI mode for each experiment.

For performing a part of the sequential test runs mentioned in the following section we also used a Sun-Blade-1000 under Solaris and an AMD Athlon under Linux. **PAXML** and **AxML** were compiled with gcc -03 on the cluster, the Sun workstation, and the Linux PC. For the compilation on the SR-8000F1 we used the native C-compiler with -03 -march=F1. On the Hitachi we decided not to use special optimization options in order to ensure comparability of **parallel fastDNAml** and **PAXML**.

### 4.2 Estimating the Expected Run Time Improvement

Since preparing and executing large tests, as with the alignments mentioned below, requires a great amount of time we propose a simple method for obtaining an estimate of the expected run time improvement of **PAXML** for a specific data/hardware combination. When the global and local rearrangement options are switched off (for details refer to the **fastDNAml** documentation) the computation of the tree is accelerated by orders of magnitude, though the quality of the final tree may decrease (the final likelihood values were only $\leq 0.5\%$ worse for the 150, 200 and 250 sequence alignments however, but this is not the sole criterion for evaluating tree quality).

In addition, as depicted in Table 1 for some tests conducted on the Sun-Blade-1000, the SEV method does not scale as good without rearrangements, such that the percentage of run time improvement can be considered as a lower bound for a big parallel run with local and global rearrangements.

Thus, the execution and comparison of the running times of **AxML** and **fastDNAml** represents a feasible approach for obtaining an estimate within reasonable time limits. Another example is given in Table 2 where we executed **AxML** and **fastDNAml** without rearrangements on a machine of the Linux cluster for predicting the run time improvement of **PAXML** vs. **parallel fastDNAml**.

Finally, this approach enables an initial analysis of hardware architecture impact on program performance and an evaluation of appropriate adaptations using the sequential program, which can then easily be integrated into the parallel version.

<table>
<thead>
<tr>
<th>data set</th>
<th>AxML with rearrangements</th>
<th>AxML without rearrangements</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 taxa</td>
<td>44.91%</td>
<td>34.13%</td>
</tr>
<tr>
<td>30 taxa</td>
<td>44.81%</td>
<td>34.34%</td>
</tr>
<tr>
<td>40 taxa</td>
<td>44.91%</td>
<td>34.75%</td>
</tr>
<tr>
<td>50 taxa</td>
<td>45.00%</td>
<td>36.04%</td>
</tr>
<tr>
<td>161 taxa</td>
<td>46.90%</td>
<td>39.48%</td>
</tr>
</tbody>
</table>

Table 1: Run time improvements for **AxML** vs. **fastDNAml** with and without the rearrangement program option on the Sun-Blade-1000
4.3 Results for Large Data Sets

For performing larger test runs with biologically significant data we extracted 3 alignments from the ARB small subunit ribosomal RNA database included in the most recent database release file (6spring2001.arb [13]) consisting of 150, 200 and 250 16S/18S rRNA sequences from organisms of the three kingdoms: Eucarya, Bacteria and Archaea. The number of distinct columns (also referred to as distinct data patterns) in those alignments ranges from 2137 up to 2330.

The overall good scalability of the optimization to the parallel program is due to the fact that the tree evaluation function is the core of the worker components, which perform the actual computation [11]. Therefore, we only included the number of executed worker processes in the tables below. The results, especially the comparison with sequential performance prediction runs, demonstrate that communication overhead is only a secondary issue for parallel performance analysis.

The large parallel tests were conducted with local and global rearrangements enabled and the quickadd option set. Earlier tests have shown [10] that there is no significant difference in run time improvement for runs with quickadd enabled and disabled.

In general the tests with larger data sets reveal two astonishing results.

Firstly, we attained run time improvements of over 60% on the cluster, which demonstrate the actual potential of our optimization in terms of floating point operation reduction, especially on relatively cheap processors with weak FPU’s.

Secondly, the tests conducted on the Hitachi SR-8000F1 confirm the significant impact of hardware architecture on the performance of PAXML. The obtained run time improvement of over 26% should not be underestimated however.

<table>
<thead>
<tr>
<th>Num. of Sequences</th>
<th>Num. of columns</th>
<th>Workers</th>
<th>Improvement</th>
<th>Estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>2137</td>
<td>8</td>
<td>62.42%</td>
<td>56.96%</td>
</tr>
<tr>
<td>200</td>
<td>2253</td>
<td>12</td>
<td>63.29%</td>
<td>57.69%</td>
</tr>
<tr>
<td>250</td>
<td>2330</td>
<td>12</td>
<td>64.60%</td>
<td>58.40%</td>
</tr>
</tbody>
</table>

Table 2: Global run time improvement PAXML vs. parallel fastDNAml on the Linux cluster

In Table 2 we present the run time improvements of PAXML over parallel fastDNAml for tests conducted on the Linux cluster, and the estimate obtained by comparing the performance of the respective sequential programs as previously described.

In Table 3 we present the results obtained on the SR8000-F1, using the specially adapted program version with split up for-loops, for analogous test runs.

<table>
<thead>
<tr>
<th>Num. of Sequences</th>
<th>Num. of columns</th>
<th>Workers</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>2137</td>
<td>14</td>
<td>26.57%</td>
</tr>
<tr>
<td>200</td>
<td>2253</td>
<td>14</td>
<td>28.52%</td>
</tr>
<tr>
<td>250</td>
<td>2330</td>
<td>14</td>
<td>28.40%</td>
</tr>
</tbody>
</table>

Table 3: Global run time improvement PAXML vs. parallel fastDNAml on the Hitachi SR8000-F1
Since \texttt{PAxML} does not implement heuristics but only a purely algorithmic optimization in all tests and on all platforms \texttt{PAxML} and parallel \texttt{fastDNAml} rendered \emph{exactly} the same output, a fact that can be verified by a simple \texttt{diff} on the output files.

Due to lack of time we were not able to perform parallel computations of larger trees. We conducted however sequential test runs in order to obtain a run time improvement estimate for a 500 taxa alignment consisting of 2751 distinct columns for demonstrating the applicability of our method to large data sets. On a single processor of the Linux cluster we measured a run time improvement of 55.51\% for \texttt{AxML}.

\section{Impact of Hardware Architecture}

Due to the significant impact of hardware architecture we include some additional figures to underline the importance of this circumstance, especially within the context of efficiently performing large phylogenetic tree computations on less complex and expensive infrastructures.

![Figure 3: Absolute execution times of PAxML and parallel fastDNAml on the Hitachi and the Linux cluster](image)

In Figure 3 we depict the absolute running times for the parallel test runs we conducted on the SR8000-F1 and the cluster. Although this comparison might seem unorthodox, due to differing numbers of workers and performance characteristics of the processors, it allows an additional interesting observation (the execution time of the 150
taxa test on the Linux cluster has been scaled to 12 worker processes). **PAXML** executes faster on the Linux cluster than on the SR8000-F1, whereas parallel **fastDNAml** in turn executes faster on the Hitachi.

In Figure 4 one can observe a similar phenomenon, for the absolute execution times of the sequential versions without rearrangements for the 150, 200 and 250 taxa alignments on the Sun-Blade-1000 and a machine of the Linux cluster. Whereas there is a big divergence in the execution times of **fastDNAml** on those architectures, the difference is marginal for **AXML**.

![Figure 4: Absolute execution times of PAXML and fastDNAml on the Sun-Blade-1000 and a machine of the Linux cluster](image)

Since the HELICS cluster at the IWR Heidelberg is equipped with 512 AMD Athlon MP 1.4GHz processors, we performed some additional sequential estimate runs without rearrangements on a comparable AMD Athlon MP 1.6GHz processor at the LRR, for assessing the impact of hardware architecture and the aptness of the cluster for large tree reconstructions. The obtained run time improvements are summarized in Table 4 and are similar to those observed on the Pentium III processors, such that we can expect an analogous performance of **PAXML** on the HELICS cluster. In addition this tests demonstrate once more the good scalability of our optimization on standard off-the-shelf processors.

Finally, we observe that, as initially stated the efficiency of our optimization, i.e. in terms of reduction of floating point operations, improves with increasing number of
distinct columns in the alignment. E.g. for “short” alignments (303 to 381 distinct columns) the sequential program rendered execution time improvements ranging between 27% and 29% on the cluster, whereas the improvement for “long” alignments (2137 to 2330 distinct columns) is well above 50%. This is due to the fact, that the length of the precalculation for-loop is constant ≤ c (see section 2), and amortizes better when the main for-loop is longer.

<table>
<thead>
<tr>
<th>Num. of Sequences</th>
<th>Num. of columns</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>2137</td>
<td>53.40%</td>
</tr>
<tr>
<td>200</td>
<td>2253</td>
<td>53.04%</td>
</tr>
<tr>
<td>250</td>
<td>2330</td>
<td>53.82%</td>
</tr>
<tr>
<td>500</td>
<td>2751</td>
<td>54.94%</td>
</tr>
</tbody>
</table>

Table 4: Global run time improvement AxML vs. fastDNAML on an AMD Athlon MP 1.6GHz

5  Availability and Current Work

The most recent distribution versions of AxML, TExML and PAXML are available for download at: http://www.bode.in.tum.de/stamatak/research.html. A PAXML version with a single binary file will soon be released.

Currently we work on the extension of 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 Distributed AxML (DAxML) and Grid AxML (GAXML). DAxML is essentially a CORBA-based derivative of PAXML, whereas GAXML is being designed as migrating grid application.

6  Conclusion and Future Work

We have presented a general method for significantly reducing the number of floating point operations and thus the execution time of the tree evaluation function for maximum likelihood-based phylogenetic tree calculations. Furthermore, the theoretical concept has efficiently been implemented in two sequential and one parallel phylogeny program. The degree of run time improvement depends on the size of the sequence alignment but also strongly on the processor architecture. We measured run time improvements ranging between 26% and 65%.

The highest run time improvements have been measured on a comparably cheap cluster of Linux PCs equipped with standard hardware and software. Thus, our approach does not only enable a fast but also cheap, in terms of infrastructure costs, approach for the efficient determination of large phylogenetic trees, especially within the context of partial replacement of traditional supercomputers by large clusters.

Future work will cover the implementation and analysis of a different parallelization approach.
As already mentioned in section 4.2 not using the rearrangement option accelerates program execution by orders of magnitude without necessarily decreasing the quality of the final tree significantly, although this has to be further investigated. Since the algorithm uses heuristics, one has to be careful about considering a final tree as "the" tree for that specific alignment. Therefore, one should rather consider methods for extracting useful information from a set of good trees. Another important factor influencing the final output is the input order of the sequences, such that fastDNAml should be repeatedly executed with various input order permutations [9].

The above observations enable a different parallelization approach. Since tree quality does not seem to decrease significantly when the rearrangement option is not set, one could attempt to obtain a set of "good" trees by distributing a large number of sequence permutations instead of topologies among workers, since a complete tree computation is significantly faster in this case. Furthermore, methods and criteria for the intelligent generation of input sequence permutations have to be established, such as to improve final tree quality, especially within the context of computations without rearrangements. Finally, algorithms and concepts are required for extracting and using information from the such obtained set of "good" trees.

In addition to this approach we will investigate in more detail the impact of processor architecture on the performance of PAxML and intend to find solutions for more efficiently exploiting the potential of the SEV method on more elaborate architectures.

Finally, we plan to parallelize ATExML based on the experiences gained during development and evaluation of PAxML.

7 Acknowledgments

We are particularly grateful to Markus Pögel from the chair for "Computer Science in Engineering, Science, and Numerical Programming" at the TUM for providing us access to the newly installed Linux cluster and for his technical support.

Furthermore, we would like to thank Ralf Ebner from the LRZ for the technical support and useful information he provided us on the Hitachi SR8000-F1.

References


