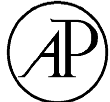

ACADEMIC PRESS

Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE @ DIRECT®

J. Parallel Distrib. Comput. 63 (2003) 707–718

Journal of Parallel and Distributed Computing

http://www.elsevier.com/locate/jpdc

# Parallel algorithms for Bayesian phylogenetic inference

Xizhou Feng,[a] Duncan A. Buell,[a,*] John R. Rose,[a] and Peter J. Waddell[b,c]

[a] *Department of Computer Science and Engineering, University of South Carolina, Columbia, SC 29208, USA*
[b] *Department of Statistics, University of South Carolina, Columbia, SC 29208, USA*
[c] *Department of Biological Sciences, University of South Carolina, Columbia, SC 29208, USA*

Received 4 December 2002; revised 15 April 2003

**Abstract**

The combination of a Markov chain Monte Carlo (MCMC) method with likelihood-based assessment of phylogenies is becoming a popular alternative to direct likelihood optimization. However, MCMC, like maximum likelihood, is a computationally expensive method. To approximate the posterior distribution of phylogenies, a Markov chain is constructed, using the Metropolis algorithm, such that the chain has the posterior distribution of the parameters of phylogenies as its stationary distribution.

This paper describes parallel algorithms and their MPI-based parallel implementation for MCMC-based Bayesian phylogenetic inference. Bayesian phylogenetic inference is computationally expensive both in time and in memory requirements. Our variations on MCMC and their implementation were done to permit the study of large phylogenetic problems. In our approach, we can distribute either entire chains or parts of a chain to different processors, since in current models the columns of the data are independent. Evaluations on a 32-node Beowulf cluster suggest the problem scales well. A number of important points are identified, including a superlinear speedup due to more effective cache usage and the point at which additional processors slow down the process due to communication overhead.

© 2003 Elsevier Inc. All rights reserved.

*Keywords:* Phylogenetic inference; MCMC; Distributed parallel computation

## 1. Introduction

One of the fundamental topics in molecular evolution is phylogenetic inference. The evolutionary relationship of sequences from different species is represented as a tree with edge lengths representing evolutionary divergence along different lineages. Phylogeny is important in understanding the process of biological evolution plus the diversity it generates. It is also the central principle for a taxonomic classification and comparison of organisms given evolutionary theory.

Over the past several decades, a large number of phylogenetic inference methods using molecular data have been proposed. In general, these methods can be separated into two classes: algorithm-oriented methods and optimization-based methods [22]. Like parsimony and likelihood, Bayesian phylogenetic inference is an optimization-based method that chooses posterior probability as the optimality criterion. Several Bayesian phylogenetic inference methods have been implemented using Monte Carlo Markov chain (MCMC) walks [9,11,12,15,28]. The posterior probability of the phylogeny (or how often it is sampled on a chain whose direction is governed by the likelihood surface) is often the key statistic retained.

Although MCMC-based methods do not appear to be the same as maximum likelihood (ML), it has have recently been shown [25] that they are closely related. That is, as sequences become long, the MCMC posterior of a tree, given an equal prior for all trees (as is typical), converges towards the normalized (i.e. to sum to 1) ML likelihood score of that tree. Thus, MCMC can be seen as an alternative computational strategy to ML with the long sequences now being used in phylogenetics, that is, the two methods will yield the same parameter estimates if the prior is equal for all trees. Given that likelihoods are sensitive to the model being used there is considerable interest in using the bootstrap combined with MCMC chains to assess posterior support for trees [2,26]. This may further increase the need for efficient parallelization of chains.

*Corresponding author. Fax: 803-777-3767.
*E-mail address:* buell@cse.sc.edu (D.A. Buell).

0743-7315/03/$ - see front matter © 2003 Elsevier Inc. All rights reserved.
doi:10.1016/S0743-7315(03)00079-0

Bayesian inference often requires integrating over many parameters, and MCMC is often chosen to fulfill such an objective. Essentially, MCMC is a sampling method combined with a continual search for large regions of high probability in a framework that is guaranteed to produce the correct distribution as the length of the chain increases [18]. The Metropolis–Hasting algorithm [8,16] is the classical MCMC algorithm and is used by almost all Bayesian phylogenetic inference approaches. Further improvements of MCMC are discussed in other works [6,13]. One effective improvement, called Metropolis-coupled MCMC [5] (MCMCMC or $(\mathrm{MC})^3$) was first used for phylogenetic inference by MrBayes [9], a Bayesian phylogenetic inference package that has become popular.

One major challenge of optimization-based phylogenetic inference lies in the doubly exponential growth, described by

$$N = \prod_{i=3}^{n} (2n - 5), \quad (1)$$

in the number of trees $N$ as a function of leaves $n$ [3]. This means, for example, that the tree space for 100 species will contain $1.7 \times 10^{182}$ unrooted trees or $3.4 \times 10^{184}$ rooted trees. Searching such a large tree space to find the best tree is computationally impossible (the problem itself is NP-hard). Currently, however, both industry and basic science *do* have a requirement to construct large phylogenies for thousands of species. Therefore, combining the fastest heuristic algorithms and the capability of parallel computing so as to facilitate large-scale phylogenetic inference is very desirable.

Several phylogenetic inference algorithms have been implemented in parallel, including fastDNAml [21], PAxML [20], Tree Puzzle [19], and GRAPPA [17]. The latest release of MrBayes also implemented parallel features [1]. Some of these implementations are reported to have very good scalability with the problem size (up to hundreds of sequences) and the number of computational nodes (up to 512 nodes).

Since no multivariate integration is easy, especially in a high-dimensional space, inferring phylogenies with hundreds of species using Bayesian methods can take weeks or more of compute time. As discussed below (Section 2.4), a single MCMC chain in an updated analysis of data such as that in Waddell, et al. [23], using whole mitochondrial protein sequences, would require memory beyond the capacity of a single CPU. Such problems will become more common when biologists want to analyze more complex models, give subsets of taxa with distinct parameter values, analyze longer alignments or more taxa.

In this paper, we describe an efficient and scalable parallel implementation of Bayesian phylogenetic inference using MCMC methods. With the appropriate synchronization and communication mechanisms, most problems can achieve linear speedup up to a limit at which parallelization overhead dominates the computation time. The algorithms we illustrated in this paper are similar to those of MrBayes [9]. Our evaluations (including bootstrapping experiments) show that they construct the same tree on the data set we used. Unlike MrBayes' parallel strategy, we have extended the parallel strategy to run likelihood calculations on different parts of the sequence data on different CPUs, so that more processors can be used to achieve greater speedup and/or provide increased memory space.

This paper is organized as follows: In the next section, we review the basic problem and the methods of Bayesian phylogenetic inference. The MCMC algorithms and likelihood evaluation are included in the overview to help in understanding the parallel implementation. In Section 3, we present two parallel strategies to speed up the computation. In Section 4, numerical experiments and performance analysis of the implementations are given. Finally, there are conclusions and suggested future improvements.

## 2. Bayesian phylogenetic inference

### 2.1. The problem

Given a set of $N$ species, each species being represented by an $M$-character sequence, Bayesian phylogenetic inference typically attempts to answer the following questions:

- What is the phylogenetic model (tree) that best explains the evolutionary relations among these species?
- With what probability is a particular tree expected to be correct?

In this paper, we use $X = \{x_{ij}\}$ to represent the sequences data and $\Psi = \{T, e, \theta\}$ for the phylogenetic model. In this notation, $T$ is the unweighted tree (sometimes called a tree topology), $e$ is the vector of edge lengths, and $\theta$ provides the parameters of the evolutionary model.

Most phylogenetic inferences are primarily concerned with the tree, but sometimes $\theta$ and $e$ are also of interest (for example, in inference of divergence times [23]).

### 2.2. Bayesian framework for phylogenetic inference

In the Bayesian framework, both the observed data $X$ and parameters of the phylogenetic model $\Psi = \{T, e, \theta\}$ are treated as random variables. Then the joint distribution of the data and model can be set up as

follows:

$$P(X, \Psi) = P(X|\Psi)P(\Psi). \tag{2}$$

Once the data is known, Bayesian theory can be used to compute the posterior probability of the model using

$$P(\Psi|X) = \frac{P(X|\Psi)P(\Psi)}{P(X)}. \tag{3}$$

Here, $P(X|\Psi)$ is called the likelihood (the probability of the data given the model), $P(\Psi)$ is the prior probability of the model (the unconditional probability of the model without any knowledge of the data), and $P(X)$ is the unconditional probability of the data, which can be computed by

$$P(X) = \int P(X|\Psi)P(\Psi)\,\mathrm{d}\Psi. \tag{4}$$

Since $P(X)$ is just a normalizing constant, the computation of (4) is not needed in practical inference.

The posterior probability distribution of the phylogenetic model is the current basis of Bayesian phylogenetic inference; much useful information can be obtained from this distribution. For example, the posterior probability of a phylogenetic tree $T_i$ can be written as

$$P(T_i) = \frac{\int\int P(X|T_i, e, \theta)P(T_i, e, \theta)\,\mathrm{d}e\,\mathrm{d}\theta}{P(X)} \tag{5}$$

while the posterior probability of the parameters of the evolutionary model can be computed as

$$P(\theta) = \frac{\sum_{T_i}\int P(X|T_i, e, \theta)P(T_i, e, \theta)\,\mathrm{d}e\,\mathrm{d}\theta}{P(X)}. \tag{6}$$

The choice of the prior probability is more or less subjective, but as the sequences become long enough with the bias of the prior not too large, the correct posterior probability is obtained asymptotically (as the likelihood of models based on the data dominates the equation [25,26]). At present most analyses use a flat prior, of all trees equally likely, which is assumed to be uninformative.

Once the prior probability of the model is chosen, the task of Bayesian phylogenetic inference can be divided into three subtasks:

- the calculation of the posterior probability or the likelihood of a specific phylogenetic model;
- the approximation of the posterior probability distribution of the phylogenetic models;
- the inference of the characteristics of the posterior probability distribution, such as the credibility interval of the parameters of the model or the Maximum Posterior Probability tree.

The following subsections will describe the methods to solve the first two tasks: using MCMC to approximate the posterior probability distribution of the phylogenetic models and fast likelihood evaluation using the recursive algorithm of Felsenstein and local update. The third task can be solved by many statistical methods and is outside the scope of this paper.

### 2.3. MCMC algorithms and improvement

The basic idea of the MCMC method is first to construct a Markov chain that has the space of the parameters to be estimated as its state space and the posterior distribution of the parameters as its stationary distribution. Next, simulate the chain and treat the realization as a hopefully large and representative sample from the posterior distribution of the interested parameters.

In practice, the Metropolis–Hasting algorithm is chosen when constructing such a Markov chain. When applied to phylogenetic inference, the Metropolis–Hasting algorithm [6,8,14,16] can be described as follows:

**Metropolis–Hasting algorithm.**

1. Initialize a phylogeny model $\Psi^{(0)}$; set $t = 0$.
2. Repeat 2.1–2.4:
   - 2.1. Propose a new sample $\Psi$ from $q(\cdot|\Psi^{(t)})$.
   - 2.2. Draw a random variable $u$ from the uniform distribution $U(0, 1)$.
   - 2.3. If $u \leqslant \alpha(\Psi^{(t)}, \Psi)$ then set $\Psi^{(t+1)} = \Psi$; otherwise set $\Psi^{(t+1)} = \Psi^{(t)}$.
   - 2.4. Set $t = t + 1$.

In the above algorithm, $\alpha(\Psi^{(t)}, \Psi)$ is called the acceptance probability, and its definition distinguishes different MCMC algorithms. In the original Metropolis algorithm,

$$\alpha(\Psi^{(t)}, \Psi) = \min\left(1, \frac{\pi(\Psi|X)}{\pi(\Psi^{(t)}|X)}\right). \tag{7}$$

Hasting extended the original Metropolis algorithm by allowing an asymmetric proposal probability $q(\Psi, \Psi^{(t)})$ and introduced a new transitional kernel

$$\alpha(\Psi^{(t)}, \Psi) = \min\left(1, \frac{\pi(\Psi|X)q(\Psi^{(t)}|\Psi)}{\pi(\Psi^{(t)}|X)q(\Psi|\Psi^{(t)})}\right). \tag{8}$$

The proposal probability can be in any form that satisfies $q(\cdot|M) > 0$, but the choice of $q(\cdot|M)$ may affect the convergence rate of the Markov chain.

The Metropolis–Hasting algorithm is the cornerstone of most MCMC algorithms and has proven effective in many applications. Theoretically, when the chain runs a large number of generations, the distribution of samples will approximate the posterior distribution of the

interested parameters. But some practical problems still exist for the original Metropolis–Hasting algorithm:

- the proposal step is hard to choose; a large step may result in a low acceptance rate, and a small step makes the chain move very slowly through the space being integrated;
- the chain may stop at local optima;
- the dimension of parameters of interest may change.

Several improvements have been proposed in recent years, and some have been introduced into phylogenetic inference. These include Metropolis-coupled MCMC [5], Time-reversible jump MCMC [7], multiple-try Metropolis [14], and population-based MCMC [13]. Metropolis-coupled MCMC was also called parallel tempering, exchange Monte Carlo [13], or $(MC)^3$. The idea of Metropolis-coupled MCMC is to run several chains in parallel, each chain having a different distribution $\pi_i(\Psi))$ $(i = 1, \ldots, m)$, and with index swap operations conducted in place of the temperature transition of simulated annealing methods. The chain with distribution $\pi_1(\Psi)$ is used in sampling and is called the cool chain. The other chains are used to improve the mixing of the chains and are called heated chains. The generalized Metropolis-coupled MCMC algorithm [5,6,13] is as follows:

**Metropolis-coupled MCMC algorithm.**

1. Initialize a phylogeny model $\Psi^{(0)}$; set $t = 0$.
2. Repeat 2.1–2.4:
   2.1. Draw a random variable $u$ from the uniform distribution $U(0, 1)$.
   2.2. If $u \leqslant \alpha_0$, a parallel step is conducted using MCMC algorithms.
   2.3. If $u > \alpha_0$, a swapping step is conducted by randomly choosing two chains $i$ and $j$, and an index swapping is proposed with acceptance probability $a = \min\left\{1, \frac{\pi_i(\Psi_j^{(t)})\pi_j(\Psi_i^{(t)})}{\pi_i(\Psi_i^{(t)})\pi_j(\Psi_j^{(t)})}\right\}$.
   2.4. Set $t = t + 1$.

An alternative of the above algorithm is to combine the parallel step and the swap step into one big step and conduct a swap step at every generation.

Metropolis-coupled MCMC is useful in overcoming the local optima problem, but is not helpful in improving the acceptance rate. Multiple-try Metropolis [14] can improve the acceptance rate by proposing multiple sample candidates and considering their combined effects in the acceptance probability. A further advantage comes if these multiple steps are computationally very similar so require reduced effort over most random steps. It is expected that multiple-try Metropolis may improve the convergence behavior of the chain, thus reducing the number of sample evaluations. The algorithm is described as follows:

**Multiple-try Metropolis [14].**

1. Initialize a phylogeny model $\Psi^{(0)}$; set $t = 0$.
2. Repeat 2.1–2.7:
   2.1. Draw $k$ independent proposals $(\Psi_1 \ldots \Psi_k)$ from a symmetric proposal distribution $q(\cdot|\Psi^{(t)})$.
   2.2. Choose $\Psi^\star$ from $\Psi_1 \ldots \Psi_k$ with probability proportional to $\pi(\Psi_i)$.
   2.3. Draw $k$ independent proposals $\Psi_1' \ldots \Psi_k'$ from a symmetric proposal distribution $q(\cdot|\Psi)$.
   2.4. Draw a random variable $u$ from the uniform distribution $U(0, 1)$.
   2.5. Compute $a = \frac{\pi(\Psi_1) + \ldots + \pi(\Psi_k)}{\pi(\Psi_1') + \ldots + \pi(\Psi_k')}$.
   2.6. If $u \leqslant a$ set $\Psi^{(t+1)} = \Psi^\star$; otherwise set $\Psi^{(t+1)} = \Psi^{(t)}$.
   2.7. Set $t = t + 1$.

### 2.4. Likelihood evaluation

Evaluating the likelihood of the data under a given model is a key component in Bayesian phylogenetic inference and maximum likelihood estimation. Most computation time in likelihood-based phylogenetic inference methods is spent in likelihood evaluation.

The objective is to compute the likelihood given a data set $X$ and a specific phylogenetic model $\Psi$, which includes a phylogenetic tree $T$, a vector of branch length $e$, and an evolutionary model $\theta$. The rate matrix $R = (r_{ij})$ is used to represent the substitution part of the mechanism of evolution (part of $\theta$) and $r_{ij}$ is the instantaneous substitution rate of the $i$th residue by the $j$th residue. Many assumptions may be introduced to make $R$ simple, easy to handle, and computationally fast [22]. The substitution probability or transition matrix $(P)$ over a finite period of time $\tau$, $P(\tau)$, can be obtained by the matrix exponent operation $P(\tau) = e^{R\tau}$.

If the substitution rate matrix is constant over all sites, then the likelihood can be defined [3] as

$$L(D|T, \tau^\bullet, \theta) = \prod_{u=1}^{m} \left( \sum_{a^{n+1} \ldots a^{2n+1}} \pi_{a^{2n-1}} \prod_{i=n+1}^{2n-2} P(a^i|a^{\alpha(i)}, \tau_i, \theta) \times \prod_{i=1}^{n} P(d_u^i|a^{\alpha(i)}, \tau_i, \theta) \right) \quad (9)$$

In the above equations, $\alpha(i)$ denotes the immediate ancestral node to node $i$, $a^i$ denotes the residue state at node $i$, and $d_u^i$ denotes the residues at the $u$th site of the

$i$th sequence. Generally, a logarithmic form of the likelihood is used; Felsenstein [4] proposed the following recursive algorithm to compute the likelihood.

**Felsenstein Algorithm for likelihood evaluation**

1. FOR $(k = 1; k \leqslant 2n-2; k++)$ compute $Q^{(k)} = q_{ij}^{(k)} = e^{R \cdot \tau_k}$.
2. Set $\ln L = 0$.
3. FOR $(u = 1; u \leqslant m; u++)$:
   - 3.1. Set $k = 2n - 1$.
   - 3.2. Compute $P(L_u^k|a)$ for all $a$ as follows:
     - 3.2.1. If $k$ is a leaf node, set $P(L_u^k|a) = 1$ if $a = d_u^k$; otherwise set $P(L_u^k|a) = 0$.
     - 3.2.2. If k is an internal node, compute $P(L_u^i|a)$, $P(L_u^j|a))$ for all $a$ at the children node $i, j$. Set $P(L_u^k|a) = (\sum_b P(L_u^i|b) \cdot P(b|a, \tau_i)) \cdot (\sum_c P(L_u^i|c) \cdot P(c|a, \tau_i))$.
   - 3.3. Set likelihood at site $u$, $L_u = \sum_a \pi_a P(L_u^k|a)$.
   - 3.4. $\ln L = \ln L + \ln L_u$.

*Note*: $P(b|a, \tau_i) = q_{ab}^{(i)}$.

Without any redundant computation, the above algorithm needs about

$$((2n-2)s^2 + 3s)\ \mathit{sizeof}(\mathit{double}) + nm\ \mathit{sizeof}(\mathit{char}) \approx 16ns^2 + nm$$

bytes of memory space. Also, $m(s + (n-1)(2s+1)s) \approx 2nms^2$ multiplication operations are needed in step 3. Here, $n$ is the number of species, $m$ is the length of the sequences, and $s$ is the number of residue states.

Since in most MCMC algorithms the phylogenetic model changes continuously and the change between two adjacent generations is small, if we record the nodes affected by a change in the parameter values and trace them back to the root, then only the conditional probability of those nodes appearing in the back tracing path needs to be recomputed; all other parts of the computation remain the same. We call this shortcut a local update. The local update can reduce the average number of nodes to be evaluated from $n$ to $\log(n)$. The disadvantage is that all the conditional probability values in the previous computation need to be kept in memory, increasing the memory requirement from $16ns^2 + nm$ bytes to $16ns^2 + nm + 8nms$ bytes.

If both the value of $P(L_u^k|a)$ and $\sum_b P(L_u^k|a)P(b|a, \tau_k)$ are kept, the computation in step 3.2.2 can be reduced further if only one child node has changed due to the alteration of a parameter.

When $n$ is large, local update can speed up the likelihood evaluation remarkably. For this reason, even though the original Felsenstein algorithm is memory efficient, most Bayesian inference programs adopt a local update scheme despite the extraordinary memory requirement. Furthermore, speedup can be obtained by compressing the patterns of the sequence (if there are $y$ columns of all state "a" in the data, calculate the likelihood of this once and raise it to the power $y$).

Practical phylogenetic inferences may use more complicated models, such as rate variation among sites [22], but the basic computation is the same as the Felsenstein algorithm except in some special cases [27].

### 2.5. The need for parallel Bayesian phylogenetic inference

From the above overview, it is obvious that Bayesian phylogenetic inference is a computationally intensive process. Consider a realistic problem, with $n = 200$, $m = 3500$, and $s = 20$ (amino acid sequence data), and a model allowing five different rates across sites ($k = 5$). Assume we use a Metropolis-coupled algorithm with 5 chains, each chain lasting 10,000,000 generations, and that we use a local update scheme in the implementation. Then at least on the order of $6 \times 10^9$ bytes of memory and at least on the order of $6 \times 10^{16}$ multiplication operations are needed. This is a real situation such as analyzing all the published whole mtDNA amino acid sequences of vertebrates [24]. To be competitive in analytic quality, more complicated models are desirable; this together with an exponential growth rate in the number of sequenced taxa means growing computational demands. Even now these demands exceed the ability of a single-CPU computer and require a longer computation time than is reasonable, hence the motivation for this research. A parallel implementation is required; we describe our parallel strategy in the next section.

## 3. Parallel algorithms

### 3.1. Issues of parallel Bayesian phylogenetic inference

Parallelizing Bayesian phylogenetic inference brings two advantages: speeding up the computation and providing the memory space needed for a competitive biological analysis of current data. Since generating the Markov chain accounts for the vast bulk of the computation, our parallelization will focus on parallelizing the MCMC algorithm. A single chain MCMC is essentially a sequential program, since the state at time $t+1$ is dependent on the state at time $t$. One way to parallelize Metropolis–Hasting MCMC is to parallelize the likelihood evaluation. Another is to run multiple equivalent chains and sample each one after the burn-in stage. This method may involve many random starting points, which provides the advantage of exploring the space through independent initial trajectories, but also has the danger that the burn-in stage may not always be cleared [6]. A single Metropolis-coupled MCMC run can be parallelized at the chain level, but chain-to-chain

communications are needed frequently. However, just parallelizing on the chain level will not use all the available resources, especially when memory is the limiting factor. Multiple-try Metropolis methods may reduce the whole computation by using a shorter chain to get the same result as a long chain and are also easy to parallelize. For illustrative purposes we focus below on Metropolis-coupled MCMC.

An important issue in the parallelization of Metropolis-coupled MCMC is balancing the load. This issue of load balancing comes from the fact that when a local update scheme is used, different chains may reevaluate over a different number of nodes at each step. More seriously, the local update scheme is often only available for topology and edge length changes; with parameters such as a global rate matrix changing, the likelihood needs to be evaluated across the tree, so all nodes need to be reevaluated.

Other issues that must be considered in a parallel algorithm are how to synchronize the processors, how to reduce the number of communications, and how to reduce the message length of each communication.

### 3.2. Task decomposition and assignment

From the description in Sections 2.3 and 2.4 there are two natural approaches to exploiting parallelism in Metropolis-coupled MCMC: chain-level parallelization and sub-sequence-level parallelization. Chain-level parallelization divides chains among processors; each processor is responsible for one or more chains. Subsequence-level parallelization divides the whole sequence among processors; each processor is responsible for a segment of the sequence, and communications contribute to computing the global likelihood by collecting local likelihood from all processors. Our implementations combine these two-approaches together and map the computation task of one cycle into a two-dimensional grid topology.

The processor pool is arranged as an $c \times r$ two-dimensional Cartesian grid. The data set is split into $c$ segments, and each column is assigned one segment. The chains are divided into $r$ groups, and each row is assigned one group of chains. When $c = 1$, the arrangement becomes chain-level parallel; when $r = 1$, the arrangement becomes subsequence-level parallel. Fig. 1 gives an illustration of how to map 8 chains onto a $4 \times 4$ grid, the length of the sequences being 2000.

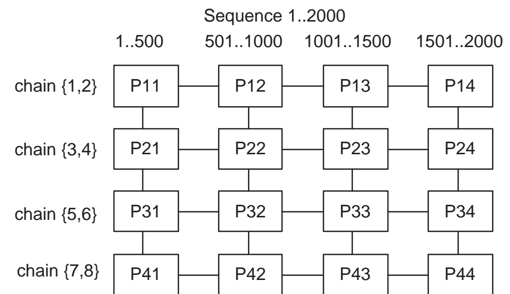

Fig. 1. Map 8 chains to a $4 \times 4$ grid, with the length of sequences being 2000.

### 3.3. Synchronization and communication

We use two sets of random number generators (RNG1 and RNG2) to synchronize the processors in the grid. RNG1 is used for intra-row synchronization. The processors on the same row have the same seed for RNG1, but different rows have a different seed for RNG1. RNG2 is used for inter-row communication. All processors in the grid topologies have the same seed for RNG2.

On each row, RNG1 is used to generate the proposal state and draw random variables from the uniform distribution. Since the same seed is used, the processors on the same row always generate the same proposal, and make the same decision on whether or not to accept the proposal. During each cycle, only one collective communication is needed to gather the global likelihood and then broadcast it to all processors on the same row, The `MPI_ALLREDUCE` function can be used to fulfill this task, each communication only needing to communicate twice as many double precision values as the number of chains on the row, that is, the local likelihood and the global likelihood. Since different rows use different seeds for RNG1, the chains on them can traverse different states.

RNG2 is used to choose which two chains should conduct a swap step and to draw the probability to decide whether or not to accept the swap operation.

When the two chosen chains are located on different rows, a two-step communication is conducted on these two rows. The first step is used to exchange the likelihood and temperature of the chosen rows. The second step is used to broadcast the result to other processors located on the grid or the chosen rows. There are several chain swap algorithms available, and we will discuss them in Section 4.

In each chain swap step, the indices but not the state information of the chains are swapped. An index swap operation changes the temperature of the chains being swapped, and the cool chain may jump from one chain to another. Index swapping reduces the communication contents needed by chain swapping to a minimum.

### 3.4. Load balancing

The processors on the same row always have a balanced load if the differences of the lengths of the subsequences on each column are small enough. The imbalance among different rows is unavoidable, since

we cannot predict the instantaneous behavior of a given chain within a step. However, some techniques can still be used to decrease the imbalance.

The first technique is to synchronize the proposal type on all chains. We use RNG2 to control how to propose a new candidate state. This can prevent the phenomenon of having one chain doing a local update and another chain doing a global update.

The second technique is to choose a swap proposal probability to control the interval between two swap steps.

### 3.5. Symmetric parallel MCMC algorithm

Until now, all the parallel strategies we have discussed are based on an assumption that any two chains chosen to conduct a swap step need to be synchronized. The whole algorithm, which we refer to as the symmetric parallel MCMC algorithm, is given here.

**Symmetric parallel MCMC algorithm**

1. Problem Initialization
    - 1.1 Init parallel environment
    - 1.2. Get resource information, number of processors, rank of processor
    - 1.3. Get problem configuration, number of chains and groups
    - 1.4. Build the topology, compute the coordinate $(r, c)$ of current processor
    - 1.5. Read in data by the head node
    - 1.6. Scatter or broadcast the data to corresponding processors
    - 1.7. Set seed for RNG1 and RNG2 (RNG: random number generator)
2. MCMC initialization
    - 2.1. Compress the sequence data
    - 2.2. For each chain on current processor
        - 2.2.1. Setup the temperature for each chain on current processor
        - 2.2.2. Set length for each branch randomly
        - 2.2.3. Choose parameters for the models
        - 2.2.4. Build a starting tree
        - 2.2.5. Compute the local likelihood using local data
    - 2.3. On each row, compute the global likelihood from the local likelihood
    - 2.4. Set $t = 0$
3. While ($t <$ maximum generations) do
    - 3.1. Draw a random variable $u_1$ from $U(0, 1)$ (use RNG2)
    - 3.2. If $u_1 \leqslant a_0$(swap probability), do swap step
        - 3.2.1. Choose chain $i$ and chain $j$ using RNG2
        - 3.2.2. Compute the coordinate $(r(i), index(i))$ of chain $i$ and $(r(j), index(j))$ of chain $j$
        - 3.2.3. If $r(i) = r(j)$ do intra-processor swap
            - 3.2.3.1. If $r = r(i)$ do
                - 3.2.3.1.1. Compute $a_s = \min\left(1, \frac{\pi_i(\Psi_j^{(t)})\pi_j(\Psi_i^{(t)})}{\pi_i(\Psi_i^{(t)})\pi_j(\Psi_j^{(t)})}\right)$
                - 3.2.3.1.2. Draw a sample $u_2$ from $U(0, 1)$ (use RNG1)
                - 3.2.3.1.3. If $u_1 \leqslant a_s$, swap the temperature and the index of chain $i$ and chain $j$
        - 3.2.4. Otherwise do inter-processor swap
            - 3.2.4.1. If $(r = r(i))$ or $(r = r(j))$ exchange the temperature and likelihood of the chain $i$ and $j$ between processor $(r(i), c)$ and $(r(j), c)$
            - 3.2.4.2. Compute $a_s = \min\left(1, \frac{\pi_i(\Psi_j^{(t)})\pi_j(\Psi_i^{(t)})}{\pi_i(\Psi_i^{(t)})\pi_j(\Psi_j^{(t)})}\right)$
                - 3.2.4.2.1. Draw $u_2$ from $U(0, 1)$ (use RNG2)
                - 3.2.4.2.2. If $u_1 \leqslant a_s$, swap the index and the temperature of chain $i$ and chain $j$ (information already got at step 3.2.4.1)
    - 3.3. Otherwise do parallel step
        - 3.3.1. For each chain on current processor
            - 3.3.1.1. Draw $u_3$ from $U(0, 1)$ (use RNG2)
            - 3.3.1.2. Choose a proposal type from $u_3$
            - 3.3.1.3. Propose a new state $\Psi$ using this proposal type
            - 3.3.1.4. Compute the local likelihood
        - 3.3.2. On each row, compute the global likelihood
        - 3.3.3. For each chain on current processor
            - 3.3.3.1. Compute $a(\Psi^{(t)}, \Psi) = \min\left(1, \frac{\pi(\Psi|D)q(\Psi^{(t)}|\Psi)}{\pi(\Psi^{(t)}|D)q(\Psi|\Psi^{(t)})}\right)$
                - 3.4.1.1. Draw $u$ from $U(0, 1)$(use RNG1)
            - 3.3.3.2. If $u \leqslant \alpha(\Psi^{(t)}, \Psi)$ then set $\Psi^{(t+1)} = \Psi$; otherwise $\Psi^{(t+1)} = \Psi^{(t)}$
    - 3.4. If sampling is needed, output the state information of the cool chain
    - 3.5. Set $t = t + 1$

### 3.6. Asymmetric MCMC algorithm

To further reduce the negative effect of imbalance between different chains, an asymmetric MCMC algorithm can be used. The idea is to introduce a processor as the coordinator node. The coordinator node is used to coordinate the communication between different rows; it does not participate in the likelihood evaluation. After each cycle, the head of each row sends the state

information of its chains to the coordinator and retrieves information from it when a swap step is proposed. The asymmetric MCMC algorithm is similar to the shared memory algorithm, but the coordinator can perform other functions such as convergence detection and sampling output.

Compared to the symmetric MCMC algorithm, the asymmetric MCMC algorithm wastes one processor. Thus, when the number of rows in the grid topology is not large, the symmetric MCMC algorithm seems favorable.

## 4. Numerical results and discussion

We implemented an MPI version of the parallel MCMC algorithms proposed in the previous section. We refer to this as the Parallel Bayesian Phylogenetic Inference (PBPI) implementation. We evaluate the implementation on a Beowulf cluster with a head node and 32 compute nodes, each node having one Intel PIII 933M CPU and 1 GB of memory. The operating system is Red Hat Linux 7.1, and the MPI used in the evaluation is MPICH 1.2.0. The following four data sets are used. These data sets are real data sets used in another of our other research project. To get the performance data, we execute 5 runs for each case, and each run lasts 5000 generations. The average execution time of the 5 runs is used to compute the speedup (Table 1).

All the evaluations use the HKY molecular substitution model with constant substitution rate, empirical base frequency, and estimated transition/tranversion ratio.

In comparison with the parallel implementation proposed in [1], our implementation provides a more generalized framework that can parallelize the Metropolis-coupled MCMC both at the chain level and the subsequence level at the same time. Moreover, our algorithm appears as presented below to scale well with larger problem sizes. This is an improvement over the recent parallel implementation of MrBayes [1], particularly as regards memory usage and in the decoupling of the number of processors used from the number of chains computed.

Table 1
Data sets used in the evaluation

| | Number of taxa | Number of characters |
|---|---|---|
| Data set 1 | 7 | 2439 |
| Data set 2 | 42 | 1031 |
| Data set 3 | 61 | 689 |
| Data set 4 | 232 | 1850 |

### 4.1. Performance measurement

The performance of a parallel algorithm is measured by speedup or efficiency. The speedup of a parallel algorithm using $p$ processors is defined as $Speedup = \frac{T_{\text{serial}}}{T_{\text{parallel}}(p)}$ and the efficiency is $Efficiency(p) = \frac{Speedup(p)}{p}$. Strictly speaking, $T_{\text{serial}}$ is the running time of the fastest known serial algorithm on one processor for the same problem. PBPI was written from scratch without further optimization and achieved approximately equal speed to that of MrBayes 2.0 when run on one processor while providing statistically the same results. So, for simplicity, we use the running time of PBPI on one processor as $T_{\text{serial}}$ in the following performance evaluations.

There are a number of possibilities for timing analysis, and different methods will give disparate results. For example, Fig. 2 shows the difference of speedup values caused by different timing methods (user time versus wall clock time) for the same example. In this paper, we chose the wall clock time, that is, the elapsed time between the start and the end of a specific run. Wall clock time will make the speedup smaller than that computed with other timing methods, such as user time. However, using wall clock time includes the cost but fails to expose the details of such negative effects as communication overhead, idle time caused by imbalance, and synchronization. One disadvantage of using the wall clock time to measure speedup is that under different computation environments, the same program may give different results if the program is not used within a dedicated environment.

### 4.2. Performance comparison of different chain swapping algorithms

Chain-to-chain swapping needs synchronization and communication between different rows in the grid topology. Reducing the overhead caused by chain

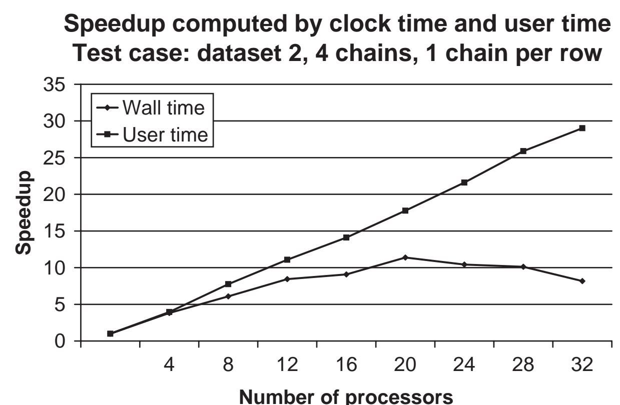


Fig. 2. Different speedup values computed by wall clock time and user time.

swapping is an important issue in parallel MCMC algorithms. We implement three chain-swapping algorithms as follows:

(1) ALG1 (synchronized algorithms): In each swap step, the likelihood of the two selected chains $I$ and $J$ are swapped by the processors with coordinates $(r(I), 0)$ and $(r(J), 0)$. The processor at $(r(I), 0)$ computes the swap ratio, decides whether the swap is taken, and then broadcasts the result to every node in the grid.
(2) ALG2 (Unsynchronized algorithms, head-to-head communication): In each swap step, the likelihoods and temperatures of the two selected chains $I$ and $J$ are swapped by the processors with coordinates $(r(I), 0)$ and $(r(J), 0)$. The processor at $(r(I), 0)$ broadcasts the swapped information to the processors at $r(I)$, and the processor at $(r(J), 0)$ broadcasts the result to the processors at $r(J)$. These processors compute the swap ratio and decide whether the swap is made. The result doesn't need to be known by the processors located at other rows.
(3) ALG3 (Unsynchronized algorithms, row-to-row communication): In each swap step, after the two chains $I$ and $J$ are selected, their likelihoods are swapped between processors at the same column, $(r(I), c)$ and $(r(J), c)$. Each of these processors computes the swap ratio and decides whether the swap is taken. No other communication is needed.

Numerical experiments indicate that ALG3 gives the best performance, then ALG2, while ALG1 is the worst (see Fig. 3). ALG3 provides 2.1 times speedup over ALG1 using 24 processors for data set 2. For data set 4, the differences of ALG2 and ALG3 are very small. But for data set 2, the performance differences between ALG2 and ALG3 are remarkable. Since ALG1 needs all rows to be synchronized at every swap step, the overhead caused by imbalance and communication in ALG1 is greatest. In ALG3, only the two selected rows need to be synchronized, using one point-to-point communication step at every swap step, so the overhead caused by imbalance among different rows can be minimized by average effects.

### 4.3. Performance of grid topology

Given a group of processors, several grid topologies can be used for task decomposition and assignment. When the number of chains is given, the grid topology can be determined by the number of chains per row and the number of all processors by

$$row_size = \frac{number_of_chains}{number_of_chains_per_row}$$

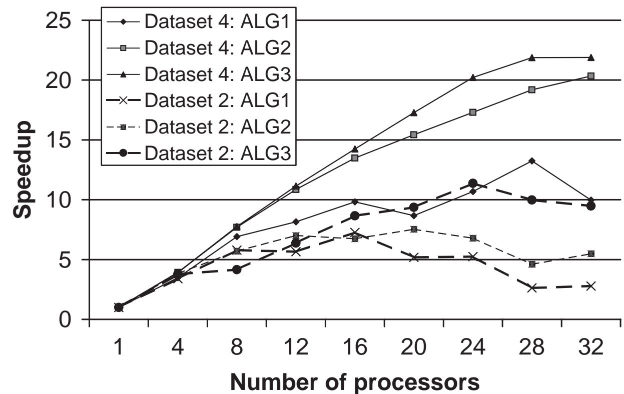

Fig. 3. Speedup for different chain-swap algorithms.

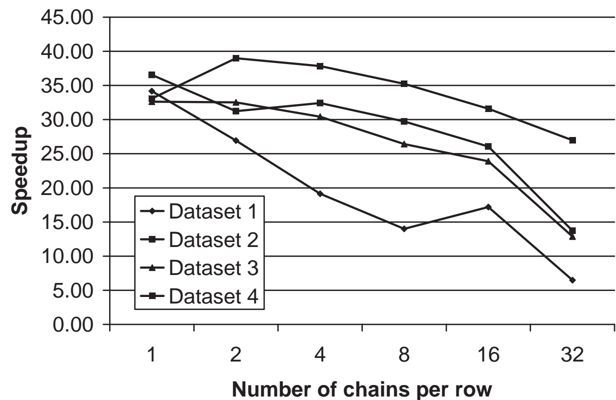

Fig. 4. Speedup of different grid topologies using ALG3.

and

$$column_size = \frac{number_of_processors}{row_size}.$$

Fig. 4 shows the speedup achieved using 32 processors running on different grid topologies for four different data sets, each data set has 32 chains. The chain-swap algorithm used here is ALG3 since it gives the best performance. The experiments show that topologies with small column size (small number of chains for each row) generally give better performance.

Fig. 4 also shows that for large problem sizes (for example, data set 4), running multiple chains (about 8 chains) on each row gives better performance than running only one chain for each row. For small problem sizes, the result is much different, and running one chain for each row always gives the best performance. This phenomenon is due to the fact that when the computation/communication ratio is high, the imbalance

between chains accounts for the major performance loss; and the communication used to collect the local likelihoods to get the global likelihood becomes the major overhead when the computation/communication ratio is low and the column size increases.

### 4.4. Scalability

The computation time of the serial MCMC algorithm (without considering memory latency in this section) is approximately

$$T_{\text{serial}} \approx \underbrace{k_1 hm \log n}_{T(localupdate)} + \underbrace{k_2 hmn}_{T(globalupdate)} + \underbrace{k_3}_{T(other)}. \quad (10)$$

Here $K_i$, for $i = 1, \ldots, 3$, is constant, $n$ is the number of taxa, $m$ is the number of character patterns, and $h$ is the number of chains.

Similarly, the computation time of the parallel MCMC algorithm can be approximated by

$$T_{\text{parallel}} \approx \underbrace{\frac{k_1 hm \log n + k_2 hmn}{p}}_{T_1} + \underbrace{k_4 \log \frac{cp}{h}}_{T_2} + \underbrace{k_5 \log \frac{h}{c}}_{T_3} + \underbrace{k_6 \log nm}_{T_4} + \underbrace{(k_3 + k_7)}_{T_5}. \quad (11)$$

Here, $K_i$, for $i = 1, \ldots, 7$, is constant, $p$ is the number of processors, and $c$ is the number of chains per row. The terms on the right-hand side of the equation represent real computation time ($T_1$), row collective communication time ($T_2$), column collective communication time ($T_3$), imbalance overhead ($T_4$), and others ($T_5$), respectively. Then the speedup of the parallel algorithm can be predicted by the following equation:

$$Speedup(p) \approx p \frac{1}{1 + \frac{p}{hm} \frac{k_4 \log \frac{cp}{h} + k_5 \log \frac{h}{c} + k_6 \log nm + (k_3 + k_7)}{k_1 \log n + k_2 n}}. \quad (12)$$

From Eq. (12), the following conclusion can be made:

- The problem size $s$ determined by $n$, $m$, and $h$. The larger the problem size, the greater the speedup obtained.
- ALG2 and ALG3 can remove the column collective communication overhead, and ALG3 further reduces row collective communication overhead. Thus, ALG3 gives the best performance.
- Communication overhead and imbalance are major factors that influence the performance scalability. For small problem size, communication overhead is important. For larger problem size, imbalance is the major obstacle.
- Assume we have synchronized the proposal types using RNG2 and also balanced the tree through tree reroot operation, then the imbalance overhead ($T_4$) is mainly caused by the sample mean of multiple random samples from $U(0, \log n)$. That means that a small exchange step will result in a larger imbalance than a large exchange step. One way to reduce imbalance overhead is to enlarge the exchange step. Another way is to use an asymmetric algorithm to decouple the interaction of different chains.
- When $p$ and $h$ are fixed, running more chains for each row may result in larger column size and then larger row communication overhead.

#### 4.4.1. Scalability with the number of processors

We tested the performance scalability with four data sets and the result is shown in Fig. 5. For data set 1 (a small data set), ALG3 can achieve linear speedup when the number of processors is less than 12. For data sets 2 and 3, linear speedup can be extended to 24 processors. For data set 4, the linear scalability can be further extended to 28 processors.

#### 4.4.2. Scalability with the problem size

The scalability of the ALG3 with the problem size is also shown in Fig. 5. On data set 4 with 4 chains we can achieve a speedup of at most 22 (using 28 processors). On data sets 2 and 3 we can achieve a speedup of at most 11 (using 20 processors). These result match the predictions from Eq. (12): Larger problem sizes lead to larger possible speedups.

#### 4.4.3. Scalability with the number of chains

Increasing the number of chains for a data set is equivalent to increasing the problem size. From Eq. (12), the direct conclusion is that using more chains will lead to greater speedup. A large number of chains can also increase the intervals between two swap steps executed on a given chain, so imbalance overhead can also be reduced. From (4), we can see that more than a 32-fold speedup can be obtained for all data sets with 32 chains. However, since MCMC algorithms sample the cool chain only, introducing too many chains leads to diminishing returns.

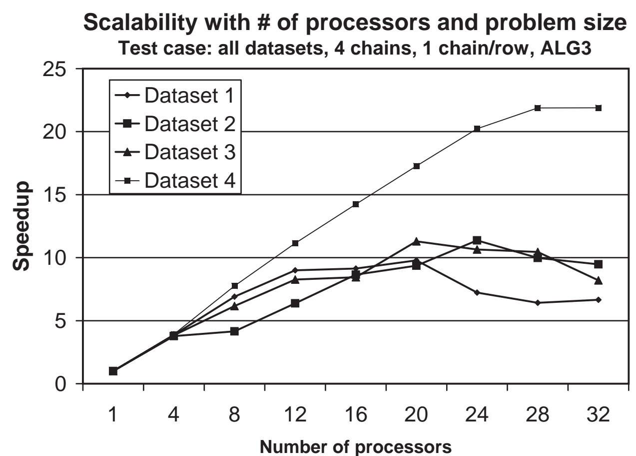


Fig. 5. Scalability with the number of processors and problem size.

## 5. Conclusions

In this paper, we described a parallel implementation of Bayesian phylogenetic inference methods using MCMC. In the proposed implementations, the processors are arranged in a 2D grid topology so that both chain-level parallel and subsequence-level parallelism can be used. We used two random number systems to synchronize the processors in the grid and reduced the overhead caused by communications and imbalances. We tested the performance of the algorithms on a 32-node Beowulf Linux cluster machines and it achieved reasonable speedup for the data sets used in the evaluation.

The algorithms proposed here scale reasonably well with the problem size. For example, we have achieved a speedup of 22 using 28 processors in one instance (data set 4 with 4 chains) and a speedup of 36 using 32 processors in another (data set 4 with 32 chains). Further, the memory space is truly distributed in our algorithms; the duplicated data is limited primarily to the input data set, and this is relatively small compared with the ongoing likelihood data. The algorithms can thus be used to infer large phylogenies that require a huge memory space and compute cycles.

In future work, we will integrate more complex phylogenetic models in our implementation and investigate the possible ways to improve the performance of MCMC algorithms using multiple-try Metropolis and population-based MCMC algorithms. Further, the general parallelization described here can be readily modified to replace the standard acceptance probabilities based on likelihood ratios, with one which uses the standard error of the difference of likelihoods between states calculated using formulae of Kishino and Hasegawa [10]. In this case the acceptance probability of a lower likelihood proposal that alters the tree equals the *p*-value of the two-tailed KH test [26] which should better reflect uncertainty when the data do not fit the model. This may avoid the high cost of the bootstrap.

## References


[1] G. Altekar, S. Dwarkadas, J.P Huelsenbeck, F. Ronquist, Parallel Metropolis-coupled Markov chain Monte Carlo for Bayesian phylogenetic inference, Technical Report TR784, Department of Computer Science, University of Rochester, July 2002.

[2] C.J. Douady, F. Delsuc, Y. Boucher, W.F. Doolittle, E.J. Douzery, Comparison of Bayesian and maximum likelihood bootstrap measures of phylogenetic reliability, Mol. Biol. Evol. 20 (2003) 248–254.

[3] R. Durbin, S. Eddy, G. Mitchison, Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids, 1st Edition, Cambridge University Press, Cambridge, 1997.

[4] J. Felsenstein, Evolutionary trees from DNA sequences: a maximum-likelihood approach, J. Mol. Evol. 17 (1981) 368–376.

[5] C.J. Geyer, Markov chain Monte Carlo maximum likelihood, in: Computing Science and Statistics: Proceedings of the 23rd Symposium Interface, 1991, pp. 156–163.

[6] W.R. Gilks, D.J. Spiegelhalter, S. Richardson, Markov Chain Monto Carlo in Practice, CRC Press, Boca Raton, FL, 1996.

[7] P.J. Green, Reversible jump Markov chain Monte Carlo computation and Bayesian model determination, Biometrika 82 (1995) 711–732.

[8] W.K. Hastings, Monte Carlo sampling methods using Markov chains and their applications, Biometrika 57 (1970) 97–109.

[9] J.P. Huelsenbeck, F. Ronquist, MRBAYES: Bayesian inference of phylogenetic trees, Bioinformatics 17 (2001) 754–755.

[10] H. Kishino, M. Hasegawa, Evaluation of the maximum likelihood estimate of the evolutionary tree topologies from DNA sequence data, and the branching order in Hominoidea, J. Mol. Evol. 29 (1989) 170–179.

[11] B. Larget, D. Simon, Markov chain Monte Carlo algorithms for the Bayesian analysis of phylogenetic trees, Mol. Biol. Evol. 16 (1999) 750–759.

[12] S. Li, D.K. Pearl, H. Doss, Phylogenetic tree construction using Markov chain Monte Carlo, J. Amer. Statist. Assoc. 95 (2000) 493–508.

[13] J.S. Liu, Monte Carlo Strategies in Scientific Computing, Springer Series in Statistics, Harvard University, New York, NY, 2001.

[14] J.S. Liu, F. Liang, W.H. Wong, The use of multiple-try method and local optimization in Metropolis sampling, J. Amer. Statist. Assoc. 95 (2000) 121–134.

[15] B. Mau, M.A. Newton, B. Larget, Bayesian phylogenetic inference via Markov Chain Monte Carlo methods, Biometrics 55 (1999) 1–12.

[16] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, Equations of state calculations by fast computing machines, J. Chem. Phys. 21 (1953) 1087–1091.

[17] B.M.E. Moret, D.A. Bader, T. Warnow, High-performance algorithm engineering for computational phylogeny, J. Supercomput. 22 (2002) 99–111.

[18] R.M. Neal, Probabilistic inference using Markov chain Monte Carlo methods, Technical Report CRG-TR-93-1, Department of Computer Science, University of Toronto, 1993.

[19] H.A. Schmidt, K. Strimmer, M. Vingron, A. von Haeseler, TREE-PUZZLE: maximum likelihood phylogenetic analysis using quartets and parallel computing, Bioinformatics 18 (2002) 502–504.

[20] A.P. Stamatakis, T. Ludwig, H. Meier, M.J. Wolf, Accelerating parallel maximum likelihood-based phylogenetic tree calculations using subtree equality vectors, in: Proceedings of the Supercomputing Conference (SC2002), Baltimore, Maryland, November 2002.

[21] C.A. Stewart, D. Hart, D.K. Berry, G.J. Olsen, E.A. Wernert, W. Fischer, Parallel implementation and performance of fastDNAml—a program for maximum likelihood phylogenetic inference, in: Proceedings of the Supercomputing Conference (SC2001), Denver, CO, November 2001.

[22] D.L. Swofford, G.J. Olsen, P.J. Waddell, D.M. Hillis, Phylogenetic inference in: D.M. Hillis, C. Moritz, B.K. Mable, (Eds.), Molecular Systematics, 2nd Edition, Sinauer & Associates, Sunderland, MA, 1996, pp. 407–514.

[23] P.J. Waddell, Y. Cao, M. Hasegawa, D.P. Mindell, Assessing the Cretaceous superordinal divergence times within birds and placental mammals using whole mitochondrial protein sequences and an extended statistical framework, Systematic Biol. 48 (1999) 119–137.

[24] P.J. Waddell, Y. Cao, J. Hauf, M. Hasegawa, Using novel phylogenetic methods to evaluate mammalian mtDNA, including AA invariant sites-LogDet plus site stripping, to detect internal conflicts in the data, with special reference to the position of hedgehog, armadillo, and elephant, Systematic Biol. 48 (1999) 31–53.

[25] P.J. Waddell, H. Kishino, R. Ota, A phylogenetic foundation for comparative mammalian genomics, Genome Informatics Ser. 12 (2001) 141–154.
[26] P.J. Waddell, H. Kishino, R. Ota, Very fast algorithms for evaluating the stability of ML and Bayesian phylogenetic trees from sequence data, Genome Informatics 13 (2002) 82–92.
[27] P.J. Waddell, D. Penny, T. Moore, Extending Hadamard conjugations to model sequence evolution with variable rates across sites, Mol. Phylogene. Evol. 8 (1997) 33–50.
[28] Z.H. Yang, B. Rannala, Bayesian phylogenetic inference using DNA sequences: a Markov chain Monte Carlo method, Mol. Biol. Evol. 14 (1997) 717–724.