2017 IEEE International Parallel and Distributed Processing Symposium Workshops

Scalable parallelization of a Markov coalescent genealogy sampler

Philip E. Davis*, Adam M. Terwilliger†‡, David Zeitler‡, Greg Wolffe†
*Rutgers Discovery Informatics Institute
Rutgers University, Piscataway, NJ, USA
†School of Computing and Information Systems
Grand Valley State University, Allendale, MI, USA
‡Department of Statistics
Grand Valley State University, Allendale, MI, USA
philip.e.davis@rutgers.edu, terwilla@mail.gvsu.edu, {zeitlerd,wolffe}@gvsu.edu

***Abstract*—Coalescent genealogy samplers are effective tools for the study of population genetics. They are used to estimate the historical parameters of a population based upon the sampling of present-day genetic information. A popular approach employs Markov chain Monte Carlo (MCMC) methods. While effective, these methods are very computationally intensive, often taking weeks to run. Although attempts have been made to leverage parallelism in an effort to reduce runtimes, they have not resulted in scalable solutions. Due to the inherently sequential nature of MCMC methods, their performance has suffered diminishing returns when applied to large-scale computing clusters. In the interests of reduced runtimes and higher quality solutions, a more sophisticated form of parallelism is required. This paper describes a novel way to apply a recently discovered generalization of MCMC for this purpose. The new approach exploits the multiple-proposal mechanism of the generalized method to enable the desired scalable parallelism while maintaining the accuracy of the original technique.**

I. Introduction

Coalescent genealogy samplers utilize stochastic statistical methods in an attempt to discover the ancestral properties of a population. They have been used to study the population genetics of HIV-1, rabbits, and bison ([1], [2], [3]). By providing significant insights into sampled populations [4], coalescent genealogy samplers can estimate various parameters of past populations such as size, growth rate, time of divergence and more ([5], [6], [7], [8]). Due to the high computational demand of coalescent genealogy samplers, parallel methods have been investigated. However, many current approaches for applying parallelism to coalescent genealogy samplers suffer from a lack of scalability due to the inherently sequential nature of the Markov chain Monte Carlo (MCMC) methods that these samplers are built upon. Therefore, achieving highly-scalable parallelism requires more sophisticated MCMC methods.

This paper demonstrates that scalable parallelism of coalescent genealogy samplers is possible through a modified version of the Generalized Metropolis-Hastings algorithm. The new algorithm allows the previously sequential initialization stage to be parallelized, removing a significant barrier to efficiently achieving a high degree of parallelism and reducing the run-time of these important tools. We propose an application, *mpcgs* (multi-proposal coalescent genealogy sampler), to implement this parallel algorithm. This paper introduces the algorithm, demonstrates its correctness in reproducing the results of the original algorithm, and characterizes the performance improvement that results from the parallel implementation.

II. Related Work

Analysis from coalescent genealogy samplers can be computationally intensive and time-consuming, due to the nature of the likelihood calculation for individual genealogies. Since a post-order traversal of the genealogical tree for each base pair position in the sequence data is required, every term must be calculated in the process of sampling each genealogical tree. Hence, the execution time of the algorithm scales linearly with sequence length, sequence count, and the number of genealogies. Numerous population-genetic software programs exist, such as BEAST [9], GeneTree [10], IM/IMa [11], MIGRATE-N [12], and LAMARC [13]. A comparison of the capabilities of each program can be found in [4]. Although these packages are useful, large-scale multi-parameter analysis with a large amount of sequence data can potentially take on the order of weeks to complete.

There has been recent progress made with respect to parallelizing multiple techniques in phylogenetics [14], [15], [16]. The application developed in this work, *mpcgs*, parallelizes a common approach to genealogy sampling: likelihood estimation. This approach utilizes MCMC, but the serial dependency of each state upon the previous state makes the parallelization of Markov chain algorithms non-trivial. A common work-around to these inherent MCMC dependencies is to execute multiple Markov chains in parallel [17], then aggregate the results of each chain [18]. Other attempts at parallelizing MCMC methods can be found in [19], [20], [21], [22].

Importantly, to ensure samples are taken from the equilibrium distribution of a Markov chain, an initial ”burn-in” phase of the chain is created and discarded. In existing parallel approaches, this burn-in phase is necessarily performed independently for each chain. This required serial burn-in phase leads to reduced efficiency at higher degrees of parallelism. In order to create efficient, highly-parallel coalescent genealogy samplers, a more sophisticated method of parallelization is required. This novel method is described in the next section, followed by implementation details and results from our application: *mpcgs*.

III. Method

The new method is a variant of the Generalized Metropolis-Hastings algorithm proposed by Calderhead [23], as applied to the coalescent genealogy sampler described in Kuhner, et al [24]. The massively-parallel computing platform used is the general-purpose graphics processing unit (GPGPU), utilizing the CUDA programming framework.

A. Generalized Metropolis-Hastings

Generalized Metropolis-Hastings, or *Calderhead's method*, differs from standard Metropolis-Hastings in that, at each iteration of the algorithm, it makes multiple proposals instead of a single proposal. That is, an implementation of Calderhead's method requires some proposal mechanism, or *proposal kernel*. The proposal kernel produces an ordered set of N new candidate states. As in the case of standard Metropolis-Hastings, there is a current state which determines the probability distribution out of which the set of proposals is drawn, but unlike standard Metropolis-Hastings, the space on which this probability distribution is defined is the set of N-tuple vectors of the states of the state space.

Calderhead presents an overview of his method in [23]; a modified version of his overview is given in Algorithm 1. The notation of $\tilde{x}_i$ represents the ith member of the proposal set, while x_i represents the ith sample taken by the method as a whole. Note that "the stationary distribution of I conditioned on the set of proposals and the state used to generate those proposals" is just the stationary distribution of the Markov chain defined by the transition matrix A. The stationary distribution of this Markov chain $P(\tilde{x}_i)$, for some member $\tilde{x}_i$ of the set of proposals, is proportional to $\pi_{\tilde{x}_j} K(\tilde{x}_j, \tilde{\mathbf{x}}_{\setminus j})$, which is the product of the density of the state in the target distribution with the probability of that state generating the rest of the current proposal set via the proposal kernel K.

Although the presence of a secondary transition matrix that guides the transition probability during the sampling stage may seem like a departure from the requirement that the transition probabilities of a Markov chain are based *solely* on the current state (and not on any other variables), this is not the case. Calderhead's method makes use of an *auxiliary variable* [25], I, which serves as the index of the set of proposals generated by the proposal kernel. Formally, the states being traversed by the method are defined by an ordered set of $N+1$ "states" of the original problem *plus*

Algorithm 1 Overview of Calderhead's Method

```
1: procedure GENERALIZED METROPOLIS-HASTINGS
2:     Initialize starting atomic state $\tilde{x}_1$, $i$ = 1, and counter $n = 0$.
3:     for each iteration do
4:         Update the set of proposals $\tilde{\mathbf{x}}_{\setminus i}$ by drawing $N$ new points from a proposal kernel, call this $K(\tilde{x}_i, \cdot)$
5:         Calculate the stationary distribution of $I$ conditioned on the set of proposals and the state used to generate those proposals.
6:         for $m = 1$ to $N$ do
7:             Sample $i$ from the stationary distribution of $I$, setting the sample $x_n + m = \tilde{x}_i$.
8:         $n = n + N$
```

some value of I. The method as a whole then is a *mixture* [25] of two different types of transitions. The first transition is the proposal stage, which changes the set of proposals. The second is the sampling stage, which changes the value of I. Additionally, at each sampling step, the sampling output is taken to be the member of the proposal set indexed by I. It is important to realize that this method is not truly transitioning from proposal to proposal, but is rather transitioning between more complex states and translating these more complex states into single proposals at each sampling step.

This method provides several opportunities for improved parallelism over a standard implementation of Metropolis-Hastings. First, the method of proposal generation can be parallelized. At the step in which the N proposals are generated, the only state information required is that of the current state, $\tilde{x}_{\setminus I}$. Therefore, the proposal kernel is dependent only upon the current state, $\tilde{x}_{\setminus I}$. While that is a liability to parallelism in standard Metropolis-Hastings, in Calderhead's method each proposal can be generated independently from all others. This allows each parallel process to separately generate a proposal. In other words, there is no interdependency between the processes generating the individual proposals. Additionally, most of the computation involved in the construction of the transition matrix can be parallelized. Since the matrix is $(N+1) \times (N+1)$ in size, each processor can populate one row of the matrix. For coalescent genealogy samplers, these two phases — proposal and sampling — account for the vast bulk of the computation. The effect is to convert a necessarily serial method to one that is very readily parallelizable. Note also that there is no distinction between the parallelism applied to the burn-in phase and that of the sampling phase. In contrast to other methods [18], there is not a necessarily-sequential burn-in component.

Developing a proposal kernel that can generate multiple proposals which can be sampled is essential. Recall that the stationary distribution of the Markov chain defined by transition matrix A is proportional to $\pi_{\tilde{x}_j} K(\tilde{x}_j, \tilde{\mathbf{x}}_{\setminus j})$ for

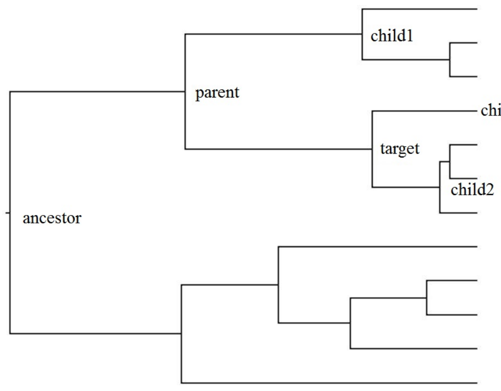

Figure 1: A tree with the neighborhood of resimulation labeled

any $\tilde{x}_j$ in the proposal set. This means that for it to be possible for a particular proposal $\tilde{x}_j$ to be sampled, it must be *possible* for the multiple proposal kernel to propose the rest of the proposal, to account for the situation that $\tilde{x}_j$ is the current state at the start of the proposal phase. In other words, for $\tilde{x}_j$ to possibly be sampled, it must be the case that $K(\tilde{x}_j, \tilde{\mathbf{x}}_{\setminus j}) > 0$.

B. The LAMARC Package

The method described here involves a modification to the software package LAMARC (Likelihood Analysis with Metropolis Algorithm using Random Coalescence) [13], which is motivated from [24] on coalescent genealogy samplers. It uses the Wright-Fischer model in estimating the parameter θ, the product of the mutation rate μ and the population size N. This approximation is found by iteratively performing an Expectation-Maximization (EM) algorithm for the likelihood of θ through Markov sampling of genealogies.

The implementation of the expectation-generation step of the EM algorithm is modified in this work. The standard proposal mechanism of LAMARC is to target one of the non-root interior nodes (the *target* node) at random from the current genealogical tree and resimulate the neighborhood around that node in such a way that proposals are selected from the distribution $P(G|\theta)$. See Figure 1 for an example of a genealogical tree with the target, parent, and child nodes labeled. Recall that each interior node represents a coalescent event. Resimulating the neighborhood consists of replacing the two deleted nodes, and possibly reordering the children of those nodes, in accordance with population genetics and coalescent theory [26].

Resimulating proportionally to $P(G|\theta)$ is achieved by treating the genealogical tree as a series of time intervals in order to determine time intervals in which the coalescent events can legally be placed probabilistically, and then placing them probabilistically inside those intervals. Figure 2 shows the tree from Figure 1 that is being resimulated. The

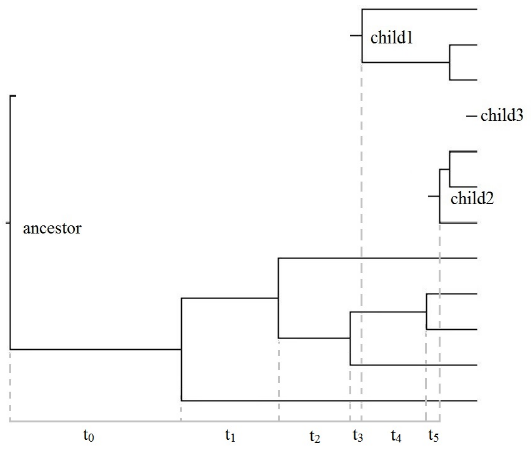

Figure 2: A tree with the neighborhood of resimulation deleted, intervals marked

lineages that have been removed and must be resimulated are called *active lineages*. At the beginning of the region being resimulated, there is a single active lineage. This lineage descends from the *ancestor* node, which is the parent of the parent of the targeted node. The first task in resimulation is to establish where coalescent events occurred. At each coalescent event, one active lineage splits into two. The earliest a coalescent event could occur is immediately following the ancestor. The times of coalescent events are bound by the location of the children in the region being resimulated. At each child, an active lineage terminates, so coalescent events must occur before the children that descend from them. Any interval that could contain a coalescent event is a *feasible interval*. In Figure 2, there are six feasible intervals. The significance of these as defined intervals is that the same number of lineages are present for the duration of the interval, which is relevant to the resimulation of the neighborhood.

The method of finding the interval in which the active lineages coalesce (i.e. the interval contains a coalescent event) is to calculate the probability of zero, one, or two active lineages coalescing for each interval, and then work backwards from the knowledge that there is exactly one active lineage at the end of the last feasible interval for coalescent placement. Suppose that higher-numbered intervals are chronologically later than lower-numbered intervals, such that interval $i+1$ is the interval immediately following interval i. Further, suppose that $P_i(n)$ is the probability that there are n active lineages at the start of interval i and that $S_{i,j}(t)$ is the probability of starting an interval of length t with i active lineages, but finishing with j active lineages. As an example, $S_{2,3}(1.0)$ is the probability of a single coalescent event happening in an interval of time length 1.0 which ends with three active lineages.

The probability function $S_{i,j}(t)$ can be derived from

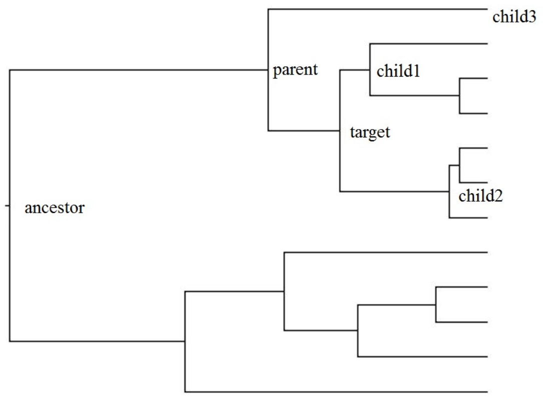

Figure 3: A possible resimulation of Figure 1

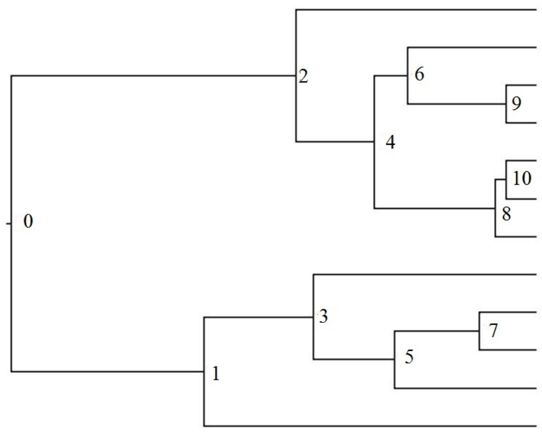

Figure 4: A tree with node labels

Kingman's coalescent theory [26] based upon a constant chance of coalescence of two active lineages, integrated over the length of the interval. This constant chance of coalescence is a function of the number of active lineages, the number of inactive lineages (lineages that are not being resimulated) and the parameter θ. Given the way intervals have been defined, there are a fixed number of inactive lineages throughout the interval. These probabilities can be used to continue backwards through the feasible intervals, populating the values of $P_i(n)$, until the ancestor node is reached.

Once the probabilities of $P_i(n)$ have been populated for all feasible intervals, the genealogical tree can then be resimulated starting from the earliest feasible interval, with the knowledge that there is exactly one active lineage at the beginning of that interval. This is accomplished by a walk forward in time, probabilistically deciding if the next interval has zero, one, or two coalescent events weighted by the values of $P_i(n)$. Once the interval in which all coalescent events will occur has been selected, the probabilistic distribution of placement within an interval is selected by treating $S_{i,j}(t)$ as a cumulative distribution function of the density of the coalescent. Additionally, the proposal may rearrange the children of the original target and its parent in order to change the structure of the overall tree [24].

The end result is a proposal which is a candidate successor genealogy. It is a modification at one interval of the current genealogical tree. See Figure 3 for an example of how Figure 1 could have been resimulated. Notice that the children have been reshuffled, changing the structure of the tree, not just the timing of the coalescent events. In this way, proposals are generated out of a distribution that is proportional to $P(G|\theta)$, where G is the proposed genealogy. Following the Metropolis-Hastings algorithm [27], the acceptance ratio necessary to sample from the posterior distribution $P(G|D, \theta)$ is $\frac{P(D|G)}{P(D|G_0)}$. The proposal is accepted with probability $min(1, r)$.

C. Modifying Calderhead's Algorithm for LAMARC

There is a challenge in attempting to apply Calderhead's algorithm using LAMARC's proposal mechanism. If two genealogies vary by more than one neighborhood, they cannot mutually propose each other. This means that Calderhead's method cannot be applied directly by simply sampling N times using the standard LAMARC mechanism, as this has a high probability of producing proposals that cannot mutually propose some other member of the proposal set. For example, consider the tree in Figure 4. If one proposal, $\tilde{x}_i$, is generated by resimulating the neighborhood around target node 6 and another proposal in the proposal set, $\tilde{x}_j$, is generated by resimulating the neighborhood around any other node, then the two proposals will vary by more than one neighborhood. There is no way that the standard proposal mechanism of LAMARC will be able to propose $\tilde{x}_j$ as a successor to $\tilde{x}_i$, or vice versa. In fact, any other proposal in the set will vary by more than one neighborhood from $\tilde{x}_i$, $\tilde{x}_j$, or both. This means that if *any* two proposals vary by more than one neighborhood, then no member of the proposal set, other than the original generator of the set, could mutually propose the rest of the proposal set. Hence, no other member of the proposal set can be sampled, which is pathological.

The method created for this research avoids the problem by using a modified proposal mechanism. The new proposal mechanism maintains an auxiliary variable, φ, which is sampled from a uniform distribution of $1 : N$, where N is the number of interior nodes in the genealogical tree. The value of φ is sampled prior to each proposal set being generated, and determines which neighborhood is targeted for resimulation.

Because φ is picked from a uniform distribution (uninformed by state information from the Markov chain) it is trivially invariant. The impact of this variable is to reduce to zero the probability of a generator state generating a set of proposals where any two proposals vary by more than one neighborhood. Any remaining set has its probability of being proposed reduced by a factor of N, however the probability distribution of these sets remains proportional to the distribution in the original LAMARC proposal mechanism. This means that the target distribution of the method remains the same, preserving correctness. This modification ensures that all proposals will be able to mutually propose the entire set, guaranteeing a non-zero sampling probability during the application of Calderhead's method.

Instead of a single acceptance ratio, Calderhead's method computes a stationary distribution on the proposal set, and samples states directly from that distribution. Recall that, in terms of coalescent genealogy samplers, the distribution is proportional to $\pi_{\tilde{G}_i} K(\tilde{G}_i, \tilde{\mathbf{G}}_{\setminus i})$ for any $\tilde{G}_i$ in the proposal set, where $\pi_{\tilde{G}_i}$ is the posterior probability of the proposal and $K(\tilde{G}_i, \tilde{\mathbf{G}}_{\setminus i})$ is the probability of $\tilde{G}_i$ mututally proposing the rest of the proposal set. These terms are calculated as follows:

$$\pi_{\tilde{G}_i} = P(\tilde{G}_i|D,\theta) = \frac{P(D|\tilde{G}_i)P(\tilde{G}_i|\theta)}{P(D|\theta)}$$

and

$$K(\tilde{G}_i, \tilde{\mathbf{G}}_{\setminus i}) = \frac{\prod_j^{N+1} P(\tilde{G}_j|\theta)}{P(\tilde{G}_i|\theta)} \propto \frac{1}{P(\tilde{G}_i|\theta)}.$$

$$\begin{aligned} \pi_{\tilde{G}_i} K(\tilde{G}_i, \tilde{\mathbf{G}}_{\setminus i}) &\propto \Big(\frac{P(D|\tilde{G}_i)P(\tilde{G}_i|\theta)}{P(D|\theta)}\Big)\frac{1}{P(\tilde{G}_i|\theta)} \\ &= \frac{P(D|\tilde{G}_i)}{P(D|\theta)} \propto P(D|\tilde{G}_i) \end{aligned}$$

Given a discrete set of proposals, this distribution is easily sampled from by sampling a real number, x, uniformly from the interval $(0, \sum_{i=1}^{N+1} P(D|\tilde{G}_i))$, and iterating through the proposals until reaching the lowest j such that $\sum_{i=1}^{j} P(D|\tilde{G}_i) \geq x$. The hidden variable I then takes value j, and the Markov chain samples the genealogy $\tilde{G}_j$. This sampling occurs an arbitrary number of times before generating a new sample set out of the distribution $K(\tilde{G}_i, \tilde{\mathbf{G}}_{\setminus i})$.

IV. Implementation

A. Program Flow

The main flow of the program is given in Figure 5. This section provides implementation details of the program, expanding on the figure as necessary.

1) Program Entry: The proposed application is called *mpcgs* (multi-proposal coalescent genealogy sampler). See https://github.com/philip-davis/mpcgs for the current version of the source code.

The command-line arguments that *mpcgs* requires at initialization are the name of a file that contains the formatted sequence data and an initial estimate of θ (θ_0.) The sequence data are expected to be in the PHYLIP genealogical data format [28], in which the first line provides the number of samples and the length of the samples. Each successive line leads with a fixed-length name of the sample followed by the sequence data. The sequence data pulled from the file specified in the first command-line argument constitutes the D term. There are no constraints on the initial estimate of θ that is provided as the second command-line argument, beyond being positive: the method as a whole is designed to be insensitive to the initial driving value of θ.

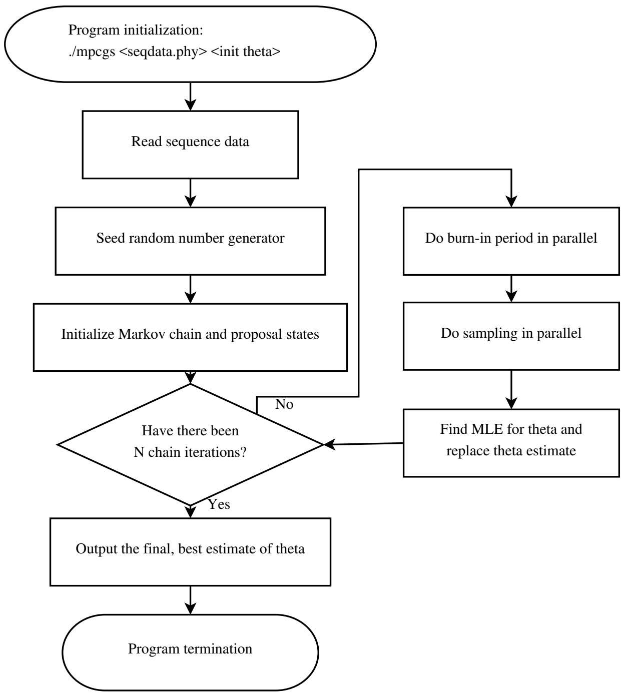
Program initialization:
./mpcgs <seqdata.phy> <init theta>
Read sequence data
Seed random number generator
Initialize Markov chain and proposal states
Have there been N chain iterations?
No
Yes
Do burn-in period in parallel
Do sampling in parallel
Find MLE for theta and replace theta estimate
Output the final, best estimate of theta
Program termination

Figure 5: The overall flow of the program. The main loop runs N times, performing N iterations of the Expectation-Maximization algorithm. N is statically defined.

2) Pseudo-Random Number Generator: There are two *pseudo-random number generators* (PRNGs) utilized in the program. The first is MT19937, a common variant of the Mersenne Twister [29]. The state for this PRNG is maintained on the host, and it is intended for random number generation by the CPU. This PRNG is used primarily to sample the auxiliary value φ from a discrete uniform distribution. However, the PRNG used by the host cannot conveniently be used by the CUDA kernels to generate random numbers since the state is held in the memory of the host (as opposed to the memory on the GPU device). Additionally, the standard deployments of MT19937 are not yet optimized for use by a GPGPU. It is therefore necessary to have a second PRNG, with its state held in the memory of the graphics card. An implementation exists of Mersenne Twister for CUDA, MTGP32, based upon [30]. This implementation maintains state for up to 256 separate threads simultaneously, and is thread-aware when run inside a CUDA kernel. This means that calls from different threads keep their state independently, with a goal of zero correlation between the numbers generated for different threads at the same point in execution.

3) Data Initialization: During the initialization stage, all memory that will be necessary over the course of the program's execution is preallocated, values that are invariant across the execution of the program are populated in device memory, the initial tree G_0 is generated and the term $P(D|G)$ is calculated.

The data structures that are preallocated include a placeholder proposal set for performing Calderhead's method. This set consists of $N + 1$ genealogical tree structures, where N is the number of proposals that will be made in the proposal stage. The preallocated data structures also include space for M genealogies that have been reduced to an array of time-intervals, where M is the number of genealogies sampled for MLE calculation. Nothing more than the time intervals are stored for each sample, since the time intervals between the coalescent events are all that is necessary to calculate the $P(G|\theta)$ term; i.e. the structure of the genealogy is not needed. Additionally, since memory allocation inside a CUDA kernel is much less efficient than allocations performed remotely on the host, it is advantageous to allocate memory for intermediate calculations as "scratch" space at this stage, rather than to continually allocate and deallocate space at run-time.

The CUDA platform provides Read-Only memory specifically for constants that will not change over the lifetime of a CUDA kernel. This constant space is highly optimized for read access. For the coalescent genealogy sampler, the data that will not change over the lifetime of execution are the sequence data. It is therefore desirable that this information be stored in constant memory for improved performance. Constant memory has optimal performance when every thread in a warp is reading the same value from constant space at the same time. In all versions of the CUDA platform, there are 32 threads in each warp. There are only four possible values that each base pair position in each sequence can take (A, C, G, or T), so each base-pair position in each sequence can be stored in two bits. The *data likelihood kernel* is designed such that each thread holds the value of a single position in a single sequence, so an entire warp can be populated out of 64 bits of data. This means that each thread in the warp can read the same 8-byte value out of constant space in order to optimize the reading of sequence data in the kernel.

Following Kuhner et al [24], *mpcgs* initializes the initial starting genealogy of the Markov chain, G_0, to be the UPGMA tree [28] generated by the distance between sequences in D. A UPGMA tree is built by clustering leafs and sub-trees according to some distance measure. The distance between individual sequences is taken to be the number of base pair positions that are different between the two sequences, while the distance between two subtrees is the arithmetic mean

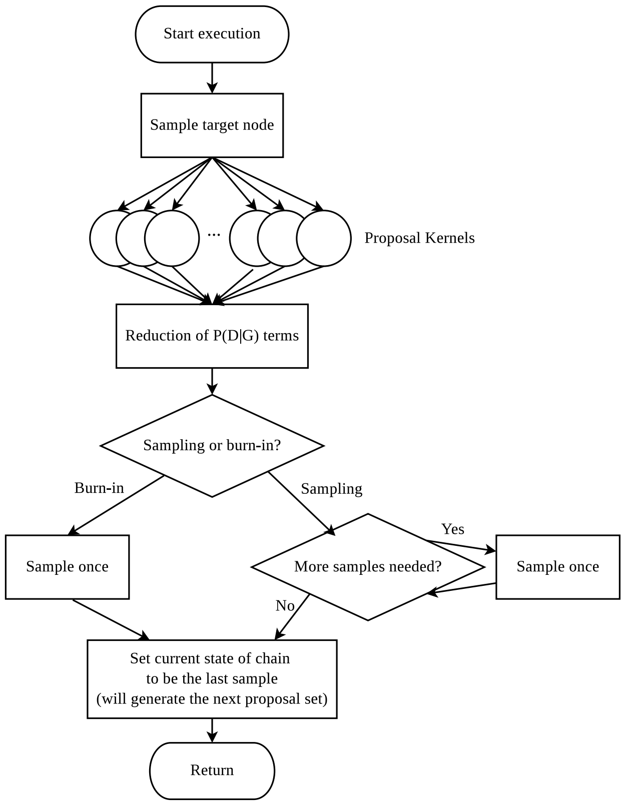

Figure 6: The multiple proposal and sampling workflow

of the distances between all the leafs across the two trees. Shorter branch lengths mean closer relations. As a deviation from the standard application of UPGMA, the branch lengths are scaled by the assumed driving value of θ. The Markov chain is then seeded with G_0 as the generating genealogy, which accomplishes step 2 of Algorithm 1.

4) Sampling by Calderhead's Method: Once the data initialization stage is complete, the Markov chain can be run using Calderhead's method. Running the chain involves two distinct periods: the burn-in period and the sample generation period. Both of these periods are realized by repeated iteration of Algorithm 1, with the addition that φ is sampled prior to each proposal set generation and passed to the CUDA kernel (called the proposal kernel) that produces proposal genealogies. The result of executing the proposal kernel is the generation of a proposal set, each member of which has a non-zero probability of proposing the rest of the set, given the value of φ. The proposal kernel additionally calculates the terms of the stationary distribution that allows sampling of the proposal set. The flow of this process can be seen in Figure 6.

5) Maximum Likelihood Estimation: Once sampling is complete, the program searches for the value of θ that has the greatest relative likelihood. This is accomplished through an iterative gradient ascent of the relative likelihood curve, the basic method of which is shown in Algorithm 2. The value of $L(\theta)$ is computed using another CUDA kernel, the *posterior likelihood kernel*.

Algorithm 2 Gradient Ascent

```
 1: set $\delta, \epsilon$ very small
 2: $\theta_{next} = \theta_0$
 3: do
 4:     $\theta = \theta_{next}$
 5:     gradient $= \frac{L(\theta_{next}+\delta)-L(\theta_{next}-\delta)}{2\delta}$
 6:     while $(L(\theta_{next}) - L(\theta_{next}+$gradient$)) > \delta$ or
        $(\theta_{next}+$gradient $) < 0$ do
 7:         gradient = gradient $\times \frac{1}{2}$
 8:     $\theta_{next} = \theta_{next}+$ gradient
 9: while $|\theta - \theta_{next}| > \epsilon$
10: $\theta_0 = \theta_{next}$
```

B. Kernels

The program described here implements three separate CUDA kernels: the proposal kernel, the data likelihood kernel, and the posterior likelihood kernel.

1) Proposal Kernel: The main difference in method from LAMARC [24] is that the neighborhood targeted for resimulation is passed to the kernel as an argument. Each thread running the proposal kernel generates one proposal genealogy. Also, each thread running the proposal kernel has the responsibility of calculating the value of $P(D|\tilde{G}_i)$, where $\tilde{G}_i$ is the proposal generated by that thread.

Since each thread produces one proposal, the total size of the generated proposal set will be equal to the number of threads executing the kernel in parallel. Each thread has an ID assigned by the CUDA runtime, and that thread ID is taken to be the index of the generated proposal in the proposal set. The thread with an ID equal to that of the current generator for Calderhead's method is idle for most of the processing, as it does not have to produce a new proposal.

All random numbers are generated prior to any possible branching of the thread execution. That is because it is a necessary condition of the MTGP32 PRNG that all threads must be executing the same random number generation request at the same time, to avoid one thread overwriting the state of another thread. The random numbers are generated prior to the resimulation of the target neighborhood, and are stored in space that was allocated during the data initialization step.

Once the proposal has been generated, each thread individually initiates an instance of the data likelihood kernel in order to determine the value of $P(D|\tilde{G}_i)$, which is the data likelihood of $\tilde{G}_i$. In order to guarantee parallel execution, a different stream of execution is initialized for each data likelihood kernel. Note that there are two layers of parallelism occurring: multiple individual proposal threads are each launching multiple child threads to calculate the data likelihood. This is an example of dynamic parallelism, supported in CUDA as the ability of threads running on the GPU device to launch new kernels. Because of the large number of total threads across all kernel executions, the load of the data likelihood calculation as a whole is shared across all the processing units of the graphics card.

After the data likelihood has been calculated, an additive reduction is performed across all the proposal threads. This is done by using the warp shuffle operators to reduce each warp down to a single value, placing that value into shared memory, and then additively reducing the values in shared memory down to a single value, a step which occurs on a single thread. Although it may seem inefficient to serially reduce this value in a single thread, in practice the number of warps will be small enough that this is not a significant portion of the total computation.

The output of this kernel is a set of proposals, as well as a discrete weighted distribution on those proposals that can be readily sampled from in order to produce sample genealogies.

2) Data Likelihood Kernel: The data likelihood kernel calculates the value of $P(D|G)$ for a given genealogy G. The D term represents the sequence data, which will be constant throughout the execution of the program. Each thread calculates the likelihood for a particular base pair position in the sequence data. In other words, each thread calculates $L^i(G) = \sum_X \pi_X L_{n_1}(X)$, where i is taken to be the thread ID.

In theory, it is only necessary to recalculate the likelihood of nodes of the tree that are the parents of branches that have changed due to the resimulation of the tree. If no aspect of a tree's structure or branch lengths has changed, then its data likelihood will not have changed either. However, in practice, the cost of uncached memory access on the CUDA platform means that it is computationally more efficient to simply recalculate the likelihood of every node in every tree for every proposal, as opposed to reading some of the values from memory. Memory accesses are required in order to find the sequence data at the tips of the tree. However, this information is stored in constant space, which decreases access times substantially.

In order to keep the results of all intermediate calculations in stack registers, the program is executed as a recursive descent through the tree, which creates one new stack frame per node in the tree per thread of execution. Unfortunately, the stack depth then varies depending on runtime variables, and there is the real possibility that a set of sequence data could overflow the stack on the CUDA device, leading to a crash. Fortunately, CUDA stack depth is configurable prior to launching a kernel, so as a solution to this problem, the program dynamically increases the size of the CUDA stack during the data initialization stage.

Once each thread has determined its value of $L^i(G)$, a multiplicative reduction must occur. Recall that the likelihoods at each base pair site are considered to be mutually independent. Just as in the reduction that occurs at the end of the proposal kernel, warp shuffle operators are used to generate one value per warp, which is placed into shared memory and then further reduced by a master thread. One thread per *block* (a higher level of thread organization than a warp) places the value into device memory. If there are enough threads to warrant multiple blocks, the execution thread that calls the data likelihood thread must perform a final reduction of the block-level aggregates. This last step is performed in serial, but the factor of reduction is so great that it does not add significantly to computation costs. The result of this kernel is the $P(D|G)$ value having been stored for the given genealogy G.

3) Posterior Likelihood Kernel: The posterior likelihood kernel calculates the value of the relative likelihood of $L(\theta)$ for a given θ and group of genealogy samples G_i, which have been previously generated by Calderhead's method. The form of $L(\theta)$ is the mean of the posterior probabilities of the members of G_i, given θ, $P(G_i|\theta)$. Each thread calculates this posterior probability term for a given genealogy in the set of samples. There are as many threads as sampled genealogies, and the thread ID is used as an index on the sample set.

Each individual thread calculates the fraction for the given genealogy. This is the main computation of the thread. Once this ratio has been computed for each genealogy of the set of samples, a reduction is performed across all threads to find the largest posterior likelihood. This provides a normalizing factor and serves to prevent value overflow. Once this normalization is complete, an additive reduction is performed on the normalized posterior values. This operation executes in the same manner as in the other two kernels.

The result of this kernel is that the expected likelihood of θ as a ratio to the likelihood of the sample-generating θ_0 has been calculated. The kernel as a whole is an implementation of the relative posterior likelihood function which is the function that is subsequently maximized in the MLE calculation in order to find a successor value of θ. This is the input into the 'Maximize' phase of the coalescent genealogy sampler, when it is viewed as an Expectation-Maximization method.

V. Results

The results consist of a demonstration of the correctness of the new algorithm in reproducing the results of the original algorithm of LAMARC, and also characterizing the performance improvement attained by the parallel implementation.

The method used to quantify correct operation was to synthesize genealogical data from a known "generating" θ value, and then compare the results of running θ-estimation using both the production LAMARC package and the proposed application, *mpcgs*.

The procedure used to synthesize genealogical data is to first generate a simulated genealogical tree using the program `ms` [31]. This program simulates the evolution of a set of "chromosomes" using a model of genetic drift.Typically, `ms` provides a set of permutations of different variants of chromosomes to represent different organisms, but it can also include the genealogical tree that was simulated to provide these results. In this case, the output of interest is the tree alone. Therefore, the command

```
ms 12 1 -T
```

was used to produce a tree in the Newick tree format [32]. The first argument specifies how many separate samples of genetic information to simulate. This will define the size of the tree. The second argument specifies how many different independent sequences of genetic information `ms` will generate for each sample.

Once a tree has been simulated, the program `seq-gen` [33] can be used to create genealogical sequence data corresponding to the tree. This program generates sequences of nucleotides which match the expected genetic separation of sequences that are related by the distances specified in the trees that `ms` generates. The command

```
seq-gen -mF84 -l 200 -s 1.0 < treefile
```

generates a set of genealogical data. The first argument (`-mF84`) specifies the model of mutation that should be used, in this case, the model is F84 (see [32] for details). The second set of arguments (`-l 200`) specifies the length of each sequence. The argument given will result in each sequence being 200 nucleotides long. The third argument (`-s 1.0`) specifies that the input generating θ for simulation purposes is 1.0. The Newick tree generated by `ms` is also given as an argument to the `seq-gen` program by input redirection. The `seq-gen` program produces sequence data in the PHYLIP format. LAMARC is packaged with utility programs to convert PHYLIP files into the necessary input format, and *mpcgs* can take these files as direct input.

The methodology was to test LAMARC and *mpcgs* with identical settings: 1,000 burn-in sample genealogies and 10,000 chain samples. Every 20th sample was used to perform gradient ascent. Each chain used the final state of the previous chain as its starting point (with the exception of the first chain, which used UPGMA). The final run consisted of 1,000 burn-in and 100,000 samples and was repeated five times.

Table I: Comparison of LAMARC and *mpcgs* with respect to θ-estimation. Average $\theta \pm \sigma$ (standard deviation) for five trials of each "generating" input θ is shown.

Input θ	LAMARC $\theta \pm \sigma$	*mpcgs* $\theta \pm \sigma$ (ours)
0.5	0.858 ± 0.024	0.966 ± 0.047
1.0	0.959 ± 0.018	1.131 ± 0.03
2.0	2.521 ± 0.064	2.423 ± 0.136
3.0	5.432 ± 0.064	5.32 ± 0.16
4.0	4.384 ± 0.067	3.913 ± 0.138

The comparison of LAMARC vs. *mpcgs* in terms of average estimated θ can be seen in Table I. Note that estimated θ values for LAMARC and *mpcgs* are not expected to be exact since these are stochastic processes. Accuracy (i.e. similarity between the methods) was tested by determining the Pearson correlation coefficient, which shows a correlation between θ-values of $r = 0.996$. Despite the differences in implementation details (such as the proposal mechanism [24]), a strong correlation is observed between the estimated θ in these two methods. Due to run-time constraints, only five trials of each generating θ per method were performed. As such, a non-parametric Friedman test for differences between the two methods was conducted and resulted in a p-value of 0.6547. Thus, we fail to reject the conclusion that there are differences between the methods. Future work will expand these analyses and explore more robust trials.

The method of testing performance improvement due to the application of parallelism (i.e. *speedup*) was to compare the runtime of LAMARC and *mpcgs* when provided the same data sets with equivalent scaling parameters. This performance characterization was made across three different dimensions of analysis that could affect performance. These were: the number of genealogical samples generated at each iteration of expectation calculation, the number of sequences in the genealogical data, and the size of the sequences (number of base pairs.)

The speedup was constant (≈ 4) with respect to the number of genealogical samples taken by each iteration of the expectation calculation phase of the EM-algorithm (or the number of samples produced by each run of the coalescent genealogy sampler). Similarly, the speedup was constant (≈ 3) relative to the number of sequences in the genealogical data being used to produce an estimate of θ. However, increasing sequence size produced scalable performance enhancements. Figure 7 demonstrates speedup increasing linearly with increased sequence length. The length of the sequence determines the amount of computation required to calculate the data likelihood for each proposal genealogy in the course of performing Calderhead's method.

VI. Conclusion

A novel method of parallelizing the coalescent genealogy sampler has been found. It allows for the parallelizaton of the burn-in period which confounded previous efforts to apply parallelism to this class of sampler algorithm. This inherently sequential component would otherwise eventually dominate computation time at higher degrees of parallelism.

With the burn-in segment parallelized, the idealized computation time for sampling becomes $\frac{B+N}{P}$, where B is the burn-in time, N is the sampling time, and P is the number of processing units on which the program is executing. Of course, in practice, there will be smaller serial components and additional implementation limits on scale, but this new method removes a significant algorithmic limitation.

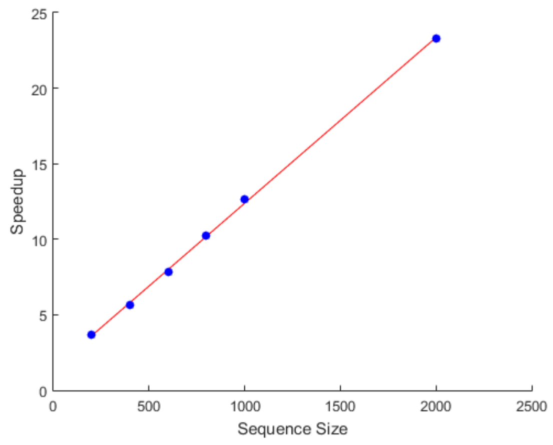

Figure 7: Speedup with varying sequence size

Applying this method to large and complex data sets using modern massively-parallel platforms should provide estimates of population parameters much faster than would otherwise be attainable.

The results also indicate that the method of implementation is most efficient at larger sequence sizes. This is fortunate, since the authors of LAMARC identify increasing the amount of genetic information per sample as generally the best way to improve study results [13]. The reason for this increased efficiency with sequence size is likely that increasing sequence size primarily increases the number of data likelihood threads executing simultaneously. This more fully utilizes the processing units of the GPU, hiding memory latency as there is a larger queue of instructions available for execution.

Overall, this method is promising for future development and scientific use. The application of Calderhead's method to create parallel coalescent genealogy samplers has the potential to greatly reduce the amount of time necessary to perform studies in population genetics by allowing the efficient application of large-scale parallelism.

References

[1] T. de Oliveira, O. G. Pybus, A. Rambaut, M. Salemi, S. Cassol, M. Ciccozzi, G. Rezza, G. C. Gattinara, R. D'arrigo, M. Amicosante *et al.*, "Molecular epidemiology: Hiv-1 and hcv sequences from libyan outbreak," *Nature*, vol. 444, no. 7121, pp. 836–837, 2006.

[2] A. Geraldes, N. Ferrand, and M. W. Nachman, "Contrasting patterns of introgression at x-linked loci across the hybrid zone between subspecies of the european rabbit (oryctolagus cuniculus)," *Genetics*, vol. 173, no. 2, pp. 919–933, 2006.

[3] B. Shapiro, A. J. Drummond, A. Rambaut, M. C. Wilson, P. E. Matheus, A. V. Sher, O. G. Pybus, M. T. P. Gilbert, I. Barnes, J. Binladen *et al.*, "Rise and fall of the beringian steppe bison," *Science*, vol. 306, no. 5701, pp. 1561–1565, 2004.

[4] M. K. Kuhner, "Coalescent genealogy samplers: windows into population history," *Trends in Ecology & Evolution*, vol. 24, no. 2, pp. 86–93, 2009.

[5] R. C. Griffiths and S. Tavare, "Sampling theory for neutral alleles in a varying environment," *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, vol. 344, no. 1310, pp. 403–410, 1994.

[6] R. C. Griffiths and P. Marjoram, "Ancestral inference from samples of dna sequences with recombination," *Journal of Computational Biology*, vol. 3, no. 4, pp. 479–502, 1996.

[7] S. M. Krone and C. Neuhauser, "Ancestral processes with selection," *Theoretical population biology*, vol. 51, no. 3, pp. 210–237, 1997.

[8] R. Griffiths and S. Tavaré, "The age of a mutation in a general coalescent tree," *Stochastic Models*, vol. 14, no. 1-2, pp. 273–295, 1998.

[9] A. J. Drummond and A. Rambaut, "Beast: Bayesian evolutionary analysis by sampling trees," *BMC evolutionary biology*, vol. 7, no. 1, p. 214, 2007.

[10] R. Page, "Genetree: comparing gene and species phylogenies using reconciled trees." *Bioinformatics*, vol. 14, no. 9, pp. 819–820, 1998.

[11] R. Nielsen and J. Wakeley, "Distinguishing migration from isolation: a markov chain monte carlo approach," *Genetics*, vol. 158, no. 2, pp. 885–896, 2001.

[12] P. Beerli and J. Felsenstein, "Maximum likelihood estimation of a migration matrix and effective population sizes in n subpopulations by using a coalescent approach," *Proceedings of the National Academy of Sciences*, vol. 98, no. 8, pp. 4563–4568, 2001.

[13] M. K. Kuhner, "Lamarc 2.0: maximum likelihood and bayesian estimation of population parameters," *Bioinformatics*, vol. 22, no. 6, pp. 768–770, 2006.

[14] A. M. Kozlov, C. Goll, and A. Stamatakis, "Efficient computation of the phylogenetic likelihood function on the intel mic architecture," in *Parallel & Distributed Processing Symposium Workshops (IPDPSW).* IEEE, 2014, pp. 518–527.

[15] D. Darriba, A. Aberer, T. Flouri, T. A. Heath, F. Izquierdo-Carrasco, and A. Stamatakis, "Boosting the performance of bayesian divergence time estimation with the phylogenetic likelihood library," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW).* IEEE, 2013, pp. 539–548.

[16] F. Izquierdo-Carrasco, N. Alachiotis, S. Berger, T. Flouri, S. P. Pissis, and A. Stamatakis, "A generic vectorization scheme and a gpu kernel for the phylogenetic likelihood library," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW).* IEEE, 2013, pp. 530–538.

[17] J. S. Rosenthal, "Parallel computing and monte carlo algorithms," *Far East Journal of Theoretical Statistics*, vol. 4, pp. 207–236, 2000.

[18] A. Sethuraman and J. Hey, "Ima2p–parallel mcmc and inference of ancient demography under the isolation with migration (im) model," *Molecular ecology resources*, vol. 16, no. 1, pp. 206–215, 2016.

[19] J. S. Liu, F. Liang, and W. H. Wong, "The multiple-try method and local optimization in metropolis sampling," *Journal of the American Statistical Association*, vol. 95, no. 449, pp. 121–134, 2000.

[20] R. M. Neal, "Mcmc using ensembles of states for problems with fast and slow variables such as gaussian process regression," *arXiv preprint arXiv:1101.0387*, 2011.

[21] A. E. Brockwell, "Parallel markov chain monte carlo simulation by pre-fetching," *Journal of Computational and Graphical Statistics*, vol. 15, no. 1, pp. 246–261, 2006.

[22] I. Strid, "Efficient parallelisation of metropolis–hastings algorithms using a prefetching approach," *Computational Statistics & Data Analysis*, vol. 54, no. 11, pp. 2814–2835, 2010.

[23] B. Calderhead, "A general construction for parallelizing metropolis- hastings algorithms," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 111, no. 49, p. 17408, 2014.

[24] M. K. Kuhner, J. Yamato, and J. Felsenstein, "Estimating effective population size and mutation rate from sequence data using metropolis-hastings sampling." *Genetics*, vol. 140, no. 4, pp. 1421–1430, 1995.

[25] C. Andrieu, N. de Freitas, A. Doucet, and M. Jordan, "An introduction to mcmc for machine learning," *Machine Learning*, vol. 50, no. 1-2, pp. 5–43, 2003.

[26] J. Kingman, "The coalescent," *Stocastic Prosess. Appl.*, vol. 13, pp. 235–248, 1982.

[27] W. K. Hastings, "Monte carlo sampling methods using markov chains and their applications," *Biometrika*, vol. 57, no. 1, pp. 97–109, 1970.

[28] J. Felsenstein, "Phylip: phylogenetic inference package, version 3.5 c." 1993.

[29] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[30] M. Saito and M. Matsumoto, "Variants of mersenne twister suitable for graphic processors," *ACM Transactions on Mathematical Software (TOMS)*, vol. 39, no. 2, p. 12, 2013.

[31] R. R. Hudson, "Generating samples under a wright–fisher neutral model of genetic variation," *Bioinformatics*, vol. 18, no. 2, pp. 337–338, 2002.

[32] J. Felsenstein and G. A. Churchill, "A hidden markov model approach to variation among sites in rate of evolution." *Molecular Biology and Evolution*, vol. 13, no. 1, pp. 93–104, 1996.

[33] A. Rambaut and N. C. Grass, "Seq-gen: an application for the monte carlo simulation of dna sequence evolution along phylogenetic trees," *Bioinformatics*, vol. 13, no. 3, pp. 235–238, 1997.