A Algorithmic details on the compact representation of the stochastic maps

In this section, we describe the algorithmic details of our representation of the stochastic maps.

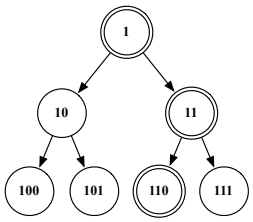
As described in Section 2.1, each of the maps F_i is completely determined by a uniform random variable U_i. Our algorithm is initialized (during the **entangle** phase of the algorithm) with two lists of uniform random numbers $V_n^{(0)}$ and $V_n^{(1)}$, $n = 1, \ldots N$, where $N = \lceil \log_2(K \times R + 1) - 1 \rceil$. These sequences should be identical in all the machines, but in practice only the seed of each of these pseudo-random sequences need to be synchronized between the machines.

Given an index $i \in \mathcal{I}$, we write its binary expansion $\mathrm{binary}(i) = (b_0, b_1, \ldots, b_n)$ to determine which items in the lists $V_n^{(b)}$ will be bitwise xored (an operation that we denote as $\oplus$), where $b_j \in \{0, 1\}$ corresponds to the j^{th} bit from the left of the binary representation of i, and $n + 1$ is the number of bits in the binary representation. b_0 denotes the most significant non-zero bit in the (leftmost non-zero bit) binary representation.

We use a binary tree to represent all of these values by denoting a root node to represent $binary(1) = \{0, \ldots, 0, 1\}$, the left child of the root to represent $binary(2) = \{0, \ldots, 1, 0\}$ and the right child of the root to represent $binary(3) = \{0, \ldots, 1, 1\}$ (see Figure 1).

The pseudocode is given in Algorithm 1, and Figure 1 shows an example of the algorithm for $i = 6$.

We performed simulations as outlined in Section 4.2. We show two random examples of pairs of runs in Figure 2, differing inside pairs only in the pseudo-random generator, and across pairs in the seed used for the generators. It can be seen that the fluctuations are negligible.



(a)

Figure 1: The representation of an implicit binary tree structure for obtaining pseudo-random numbers with $O(1)$ storage and $\log(KR)$ random access time. The double layered nodes of the tree are the nodes traversed to obtain the map for $i = 6$.

Algorithm 1 : **uniform**(i)

1: $U \leftarrow V_0^{(0)}$
2: **if** $(i = 1)$ **then**
3: return U
4: **end if**
5: $\{b_0, b_1, \ldots, b_n\} = \mathrm{binary}(i)$
6: **for** $j = 1..n$ **do**
7: $U \leftarrow U \oplus V_j^{(b_j)}$
8: **end for**
9: **return** U

B Resampling

In this section, we discuss strategies to handle cases where the number of particles is so large that even our compact representation of the genealogy might become a memory bottleneck. A related issue it that the transmission of all the individual weights can become prohibitive. These problems can be addressed by storing the genealogy in a distributed hashtable and by having each machine

communicating the sum of its weights rather than the individual weights. However, we need to first show that resampling can still be done efficiently with only the knowledge of the sum of the weights per machine.

We start by reviewing the non-parallel version of the resampling algorithm. Given the (unnormalized) weights $w_{r,1}, \ldots, w_{r,K}$, the first step is to normalize them to get the mean parameters $\mu = (\bar{w}_{r,1}, \ldots \bar{w}_{r,K})$ of a multinomial distribution $\mathrm{Mult}(K, \mu)$ over $1, \ldots, K$. Using the fact that the k-th order statistics is distributed according to a $\mathrm{Beta}(K, K-k+1)$, this can be done in time $O(K)$ using a stick breaking procedure.

The first and simplest way to do distributed resampling is to have each machine transmit the individual weight of each of its particle. This means that after this first version of **exchange** is completed for generation r, each of the machines has a global view on the weights of the entangled sampler.

The weights and the shared stochastic maps can then be used to perform a coherent resampling in all machines. This is done by letting the shared random maps correspond to particle indices, $\mathscr{G} = \{G_i : i \in \mathcal{I}\}$. Since accessing each map takes time $O(rK)$, the overall time is $O(K \log(rK))$, and the space requirements scale in $O(K)$.

Our efficient distributed resampling scheme is based on an elementary observation: if K uniform $[0,1]$ random variables are sampled, the number of points that will fall in the first half of the interval is $\mathrm{Bin}(K, 1/2)$ distributed. We use this idea recursively on a collection of dyadic intervals subdividing $[0,1]$ to get our distributed resampling scheme. More precisely, we let $\{I_j : j \in \mathcal{J}\}$ denote the dyadic intervals of size 1, $I_0 = [0,1]$, down to size $1/K$. To simplify the notation, we will assume that K is a power of two, and we will also drop the subscript r—it is understood that each resampling datastructure needs to be generation-specific.

The number of dyadic intervals is $2K-1$, which can be viewed as being organized on a tree where each interval index $j \neq 0$ has a parent index $j' = \mathrm{pi}(j)$ corresponding to the smallest distinct interval $I_{j'}$ containing I_j. We will also assume that the index of an interval corresponding to the first half of its parent interval has an odd index j, and we will denote the length of an interval I by $\ell(I)$.

For each interval that is not a leaf on the tree defined above, we attach a $(\{1, \ldots, K\} \to \{1, \ldots K\})$-valued random map induced by the artificial transition probabilities $T(x, x') = p_{\mathrm{Bin}(x,1/2)}(x')$, where $p_{\mathrm{Bin}(n,p)}$ is the probability mass function of a $\mathrm{Bin}(n,p)$-distributed random variable. These maps give us an efficient way of finding the number $N(I_j)$ of uniform random variables that fall in the first half of any non-leaf dyadic interval I_j, $j \neq 0$:

$$N(I_j, \mathscr{G}) = \begin{cases} G_{\mathrm{pi}(j)}(N(I_{\mathrm{pi}(j)}, \mathscr{G})) & \text{if } j \text{ is odd} \\ N(I_{\mathrm{pi}(j)}, \mathscr{G}) - G_{\mathrm{pi}(j)}(N(I_{\mathrm{pi}(j)}, \mathscr{G})) & \text{o.w.} \end{cases}$$

Finally, we attach to the indices at the leaves a collection of random maps described in Section 1 to be able to instantiate the location of the uniform random variables inside any leaf dyadic interval. Given an arbitrary interval $I \subset [0,1]$, it is then efficient to find how many of the K uniform random variables fall inside I: if $\ell(I) < 1/K$, then it is contained in at most two leaf dyadic intervals, while if $\ell(I) \geq 1/K$, $I \subset I'$ where I' can be written as the union of two dyadic intervals such that $\ell(I') \leq 4\ell(I)$.

Given this function N, the resampling step is computed as shown in Algorithm 2. Note that the algorithm only depends on the sum of normalized weights with indices less than or equal to the particle number k in the current generation, $\bar{w}_{\leq k} = \sum_{k' \leq k} \bar{w}_{i(r,k')}$, which only needs the sum of the weights in the other machines and the weight of the individual particles in $\mathcal{I}_r$ to be processed in the current machine.

There are $\log K$ accesses to the datastructure $\mathscr{G}$ required, for a total running time of $O(\log^2(rK) + K_m)$ and memory $O(K_m)$.

C Distributed genealogy

In this section, we discuss distributed hash table (DHT) for scaling the EMC simulation to handle the cases where the number of particles is arbitrarily large.

Algorithm 2 : **resample**$(w = w_r, \rho, \mathcal{I}_r, \mathscr{G})$

for $i \in \mathcal{I}_r$ **do**
  $k \leftarrow N([0, \bar{w}_{\leq k(i)}), \mathscr{G})$
  $n \leftarrow N([\bar{w}_{\leq k(i)}, \bar{w}_{\leq k(i)+1}, \mathscr{G})$
  for $m \in \{1, \ldots, n\}$ **do**
    $i' \leftarrow i(r+1, k+m)$
    $\rho(i') \leftarrow i$
  end for
end for

The reconstruction algorithm requires the particle genealogy to be stored in the memory. Our algorithm requires only the compact representation of the particle consisted of its stochastic map and the pointer to its parent $\rho(i)$ be stored in the memory. However, for an EMC simulation with K particles and R generations, the upper bound on the storage is $O(KR)$, which is directly proportional to the number of particles K. This is a crude upper bound, derived by assuming that all of the particles survive the resampling stage at every generation (in other words, each particle has exactly one descendent and this occurs at each generation). In fact the coalescent theory guarantees that coalescent events occur surprisingly frequently (see [1]) and in the problem domain of interest, we observed that this is indeed that case in practice.

Nonetheless, the storage of entire particle genealogy locally in a node becomes infeasible as the number of particles increases. This limitation prohibits EMC from scaling up to an arbitrarily large number of particles. To alleviate this problem, we introduce *distributed hash table* (DHT) for storing the particle genealogy.

The main idea is for each machine to track only portions of the genealogy rather than entire genealogy. Each machine stores a list of neighboring nodes, which can be queried for the parent $\rho(i)$. The neighboring node either provides the parent $\rho(i)$ or queries its neighbor list for $\rho(i)$. The idea of *chord* DHT proposed in [2] requires that each machine to keep track of at most $\log M$ other machines for routing of the queries for retrieval of parent $\rho(i)$ with at most $O(\log M)$ queries sent around the network of machines. Furthermore, join and failure of machines are handled with $O(\log^2 M)$ messages sent around the network. Since EMC relies on a network protocol for communication, being able to handle node failure is particularly useful. The use of DHT implies that the storage requirement on each machine is limited to the set of particles to be processed, $\mathcal{I}_r$, and only a subset of the genealogy of compact particles.

Defining unique key-value pair for the DHT is simple. Since any particle in the genealogy is uniquely determined by the generation r and index k through the mapping $i(r, k)$, it can be used as the key for storing and accessing any particle in the DHT. The value for each key would be the data structure containing its parent $\rho(i)$ and the stochastic map. The use of DHT comes at a cost of sacrificing speed but it allows EMC to be applied to scale up to an arbitrary number of machines and an arbitrary number of particles.

D Storage limitations

In Section C, we addressed one form of storage limitation that arises from increasing the number of particles in the EMC simulation. In this section, we discuss another form of storage limitation where the raw dataset cannot be accommodated in a single node.

The EMC algorithms in its current form focusses on cases such as phylogenetic inference or pedigree analysis where the communication between nodes is prohibitive due to the structure of the models, where it usually involves information on latent variables rather than dataset transmission. For example, the class of problems where we expect EMC to be especially attractive is when efficient SMC sampling depends on Rao-Blackwellization. In these cases the size of the particles is often increasing with respect to generation by the storage of dynamic programming tables.

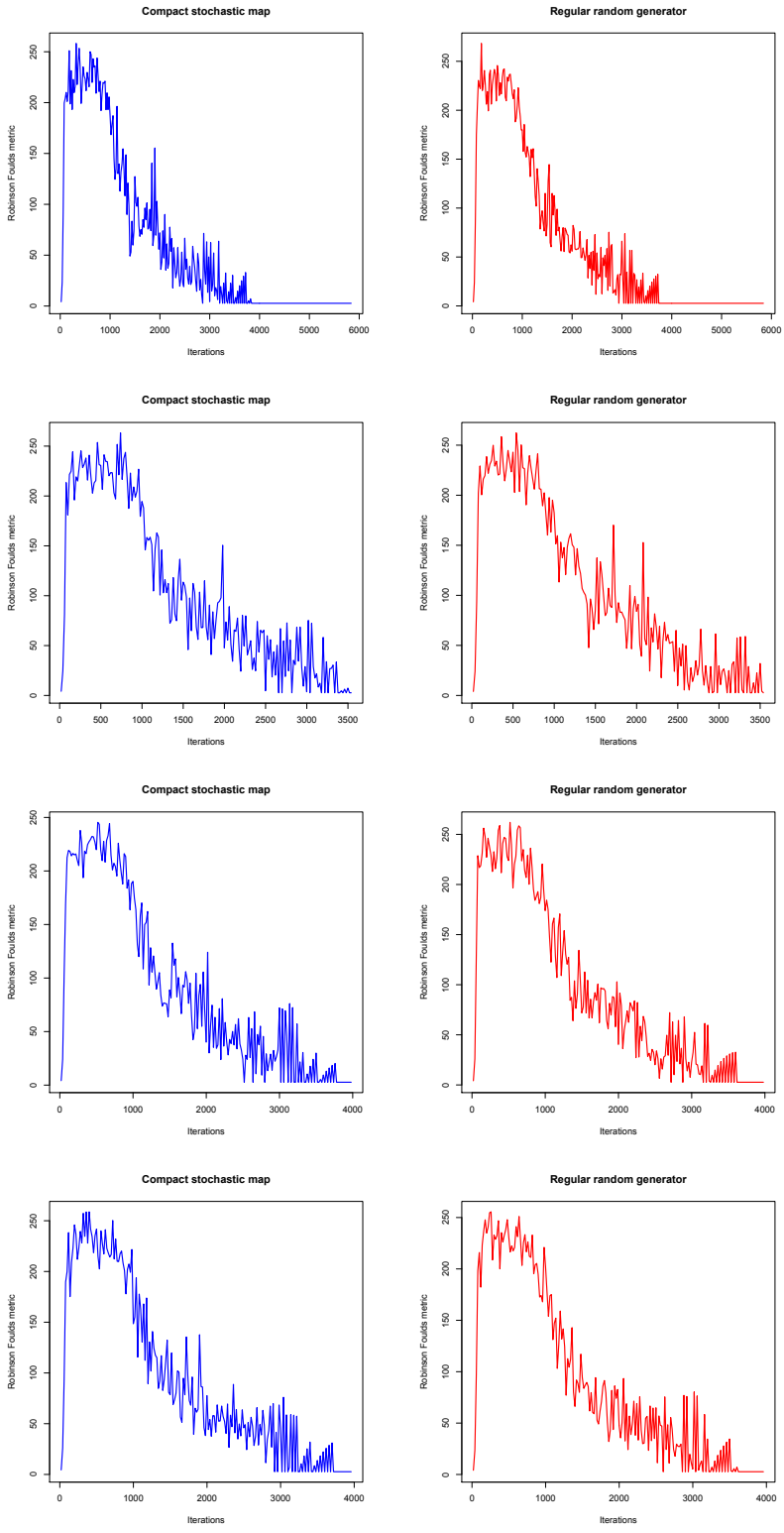
For problems involving dataset transmission, the common approach is to use MapReduce (see [3]) to distribute and process jobs in parallel. In principle, EMC can be relaxed to overcome this limitation by partitioning the data and ensuring that the allocation scheme does not cross these partition blocks.

References

[1] Joseph Felsenstein. *Inferring phylogenies*. Sinauer Associates, 2003.

[2] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM 2001*, pages 149–160, 2001.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.



(a) (b)

Figure 2: Simulations to compare the stochastic maps presented in Section A to a regular pseudo-random number generator. (a) Phylogenetic inference carried out with compact representation of stochastic maps. (b) Phylogenetic inference carried out with regular pseudo-random number generators.