Interacting Particle Inference for Probabilistic Programming in Haskell

Per Engström
Abstract

Interacting Particle Inference for Probabilistic Programming in Haskell

Per Engström

Probabilistic programming shows much promise as a declarative way to define statistical models, but inference is often expensive. A parallelisable particle Markov chain Monte Carlo sampler is implemented in Haskell and the domain-specific language Monad-Bayes. The method shows good performance compared to a single SMC sampler, but the full potential of the method could not be achieved.
# Contents

1. **Introduction**  
   1.1. Probabilistic programming ........................................ 6

2. **Theory**  
   2.1. Inference on probabilistic programs ................................ 7  
      2.1.1. SMC and CSMC ................................................ 8  
   2.2. Monad-Bayes ...................................................... 8  
   2.3. iPMCMC .......................................................... 9

3. **Method**  
   3.1. SMC and CSMC implementation .................................... 11  
   3.2. iPMCMC implementation ........................................... 12

4. **Results**  
   4.1. Parallelisation .................................................... 13  
   4.2. Correctness ....................................................... 13  
   4.3. Evaluation ......................................................... 13

5. **Conclusion**  
   5.1. Limitations ....................................................... 14  
   5.2. Future ............................................................ 15

A. **Runtime information**  
   A.1. iPMCMC .......................................................... 17  
      A.1.1. Original ...................................................... 17  
      A.1.2. Uniform node sampling ..................................... 17  
      A.1.3. Uniform trajectory sampling ................................. 18  
      A.1.4. Uniform node and trajectory sampling ....................... 19  
   A.2. Many SMC ......................................................... 19  
   A.3. SMC .............................................................. 20
1. Introduction

Bayes’ theorem

\[
\Pr(\theta \mid D) = \frac{\Pr(\theta) \Pr(D \mid \theta)}{\Pr(D)} \tag{1}
\]

about conditional probability is well-known from any introductory course on probability theory. It expresses how one’s prior believed distribution of \( \theta \) (\( \Pr(\theta) \)) is updated given observations \( D \) into a posterior belief \( \Pr(\theta \mid D) \).

The likelihood \( \Pr(D \mid \theta) \) represents the probability of generating the observation \( D \) given a particular \( \theta \). The marginal likelihood \( \Pr(D) \) acts as normalisation to ensure the distribution is valid.

The posterior distribution is valuable as it enables prediction from observations. However, unless the prior and likelihood are simple distributions then the posterior is often intractable \[2\] requiring numerical methods to sample from. One approach for approximating complicated posteriors is the family of MCMC methods \[12\]. The particle Markov chain Monte Carlo methods by Andrieu et al. \[1\], and in particular the particle Gibbs (PG) sampler, aims to improve the proposals for the MCMC sampler by using importance sampling in the form of sequential Monte Carlo (SMC) \[3\]. The PG sampler uses a modified SMC sampler conditioned on an existing particle trajectory, conditional SMC (CSMC).

The CSMC sampler is prone to path degeneracy. If the resampling collapses then, by design, only the conditional trajectory remains, reducing the mixing required for a healthy MCMC step. Rainforth et al. \[11\] proposes a solution calle Interacting Particle Markov Chain Monte Carlo (iPMCMC) where several CSMC and SMC sampler are run in parallel. By sampling the conditional trajectories possibly from independent (unconditional) SMC samplers the mixing is improved. In addition, the nodes only interact briefly promising a high degree of parallelisation.

This thesis presents a Haskell implementation of the iPMCMC sampler for Bayesian inference in the probabilistic programming DSL Monad-Bayes \[13\]. Haskell\[1\] is a general-purpose purely functional lazy programming language. To readers unfamiliar with the language an introductory text is recommended \[10\]. The monad abstraction for structured computation is well-suited for DSLs, and type classes allows flexible implementation of probabilistic functionality and inference.

1.1. Probabilistic programming

One promising way to express statistical models, and in particular Bayesian problems is probabilistic programming \[5\]. This paradigm leverages the expressiveness and familiarity of programming languages to define models independently of inference, allowing the models to be composable.

In the design of a probabilistic language the main trade-off is between expressiveness and performance of inference. A restricted language like BUGS \[4\] makes the inference simpler. In universal (Turing-complete) languages like Anglican \[14\] and Monad-Bayes \[13\] the inference is more challenging.

Central to all probabilistic languages is the ability to construct more complex models using simpler building blocks, usually primitive distributions and use Bayesian filtering. Consider the problem of regression of some complicated function \( y = f(\theta, x) \) parametrised by a parameter \( \theta > 0 \) and observations \( D = \{(x_i, y_i)\}_{i=1}^N \) influenced by some normal noise \( \delta \sim \text{Norm}(0, 1) \) such that

\[
y_i = f(\theta; x_i) + \delta. \tag{2}
\]

In Bayesian modelling, we want to find the distribution for the parameter \( \theta \) given the data \( D \), i.e. \( \Pr(\theta \mid D) \). If the function is implemented as

```haskell
reg :: MonadInfer m => [(Double, Double)] -- Observations -> m Double
reg obs = do
```

In addition, the nodes only interact briefly promising a high degree of parallelisation.

This thesis presents a Haskell implementation of the iPMCMC sampler for Bayesian inference in the probabilistic programming DSL Monad-Bayes \[13\]. Haskell\[1\] is a general-purpose purely functional lazy programming language. To readers unfamiliar with the language an introductory text is recommended \[10\]. The monad abstraction for structured computation is well-suited for DSLs, and type classes allows flexible implementation of probabilistic functionality and inference.

1.1. Probabilistic programming

One promising way to express statistical models, and in particular Bayesian problems is probabilistic programming \[5\]. This paradigm leverages the expressiveness and familiarity of programming languages to define models independently of inference, allowing the models to be composable.

In the design of a probabilistic language the main trade-off is between expressiveness and performance of inference. A restricted language like BUGS \[4\] makes the inference simpler. In universal (Turing-complete) languages like Anglican \[14\] and Monad-Bayes \[13\] the inference is more challenging.

Central to all probabilistic languages is the ability to construct more complex models using simpler building blocks, usually primitive distributions and use Bayesian filtering. Consider the problem of regression of some complicated function \( y = f(\theta, x) \) parametrised by a parameter \( \theta > 0 \) and observations \( D = \{(x_i, y_i)\}_{i=1}^N \) influenced by some normal noise \( \delta \sim \text{Norm}(0, 1) \) such that

\[
y_i = f(\theta; x_i) + \delta. \tag{2}
\]

In Bayesian modelling, we want to find the distribution for the parameter \( \theta \) given the data \( D \), i.e. \( \Pr(\theta \mid D) \). If the function is implemented as

```haskell
reg :: MonadInfer m => [(Double, Double)] -- Observations
reg obs = do
```

```haskell
1
https://www.haskell.org
```
then assuming a prior \( \Pr(\theta) \sim \Gamma(1, 1) \), the model is also a function taking the list of observations

```haskell
reg :: MonadInfer m
    => [(Double, Double)] -- Observations
    -> m Double
    reg obs = do
        theta <- gamma 1 1
        let fun = f theta
            sigma = 1
        forM obs (\ (x, y) -> do
            let mu = fun x
            p = normalPdf mu sigma y
            score p)
        return theta
```

where we for each observation score the likelihood—how probable the data is given the parameter (line 10)—skewing the prior distribution (line 7) favouring values close to where the error is small. Note how the formulation closely matches the mathematical definition. This thesis will focus on efficient but approximate sampling-based methods for finding the distribution in such problems by implementing the iPMCMC sampler and evaluating the parallelisation and comparing it to samplers in Monad-Bayes.

2. Theory

This section contains the necessary theory behind inference and probabilistic programs as well as an overview of the Monad-Bayes library and a description of the iPMCMC method.

2.1. Inference on probabilistic programs

A hidden Markov model is a model with some hidden state \( x_t \) evolving by some known process \( f(x_t \mid x_{t-1}) \) and \( \mu(x_0) \) where both functions should be seen as kernels \([6]\). In other words, the value of \( x_t \) is not known, but the underlying process is. In addition, some observations \( y_t \) are known together with their emission process \( g(y_t \mid x_t) \), see Figure 1. By comparing the measured values \( y_t \) with their distribution \( g \) and proposed hidden values \( x_t \) it is possible to score the proposals and use the scores \( w_t \) to find the most likely values.

![Figure 1: A diagram of a hidden Markov process. The hidden state \( x_t \) is given in rectangles and their observable emissions in circles.](image1)

![Figure 2: A program seen as a hidden Markov model. The emission function \( g \) defines the reported likelihood \( w \).](image2)

In the context of models expressed as probabilistic programs the time aspect is more accurately viewed as the execution of the program as seen in Figure 2. As the program executes, random variables are sampled and
This corresponds to \( f \) above. The program is interrupted at various points by scoring the current execution path, possibly depending on the sampled values. This is the likelihood score, and the method for determining the score is model-dependent on \( g \). These scorings separates the programs into parts, the \( x_t \)'s.

2.1.1. SMC and CSMC

![Figure 3: Possible execution of SMC with \( N \) particles. Each column represents a particle, and the rows the intermediary states of the program. The arrows denote the resampling heritage.]

In SMC the program is run several times. If the program is run \( N \) times it’s said to have \( N \) particles. The execution is interrupted at each scoring, and resampling is performed based on the scores \( w_t, 1:N \) according to

\[
x_t^i \sim f(x_t^i | x_{t-1}^{a_t^i})
\]  

(3)

where \( \Pr(a_{t-1}^i = \ell | w_{t-1,1:N}) = w_{t-1,\ell} \). That is, execution is not necessarily continued from where it left off, but an ancestor is chosen at random, weighted on its score. This continues until the program is completed at time \( t = T \).

A variant of SMC called conditional sequential Monte Carlo (CSMC) uses an supplied trajectory \( x_1^T \) to influence the resampling. The conditional trajectory always resamples from itself. A popular sampler using CSMC is the PG sampler. Several CSMC samplers are run in series, sampling the conditional trajectory from the surviving trajectories in the last step.

2.2. Monad-Bayes

The Haskell DSL aims to provide an universal probabilistic framework in the functional language Haskell. Monad-Bayes supports both discrete and continuous variables and it’s inference performance is comparable to that of Anglican and Prob-C. The actual library is yet to be released but is already available\(^2\). The upcoming release will be based on the simple branch and this is the version\(^3\) used in this thesis.

The simple branch differs from the implementation discussed in the paper \cite{boeing2021}. The major change is a move from generalized abstract data structures (GADT’s) to only using type classes. The relevant type classes are MonadSample for monads supporting drawing random values and MonadCond for monads supporting scoring execution paths. In addition a third type class MonadInfer exists for monads supporting both.

The MonadSample type class includes methods for drawing values from several continuous and discrete distributions but the minimal implementation is a single function

\[
\text{class Monad } m \Rightarrow \text{MonadCond } m \text{ where}
\]

\[
\begin{aligned}
\text{score} & \colon \text{Log Double} \rightarrow m ()
\end{aligned}
\]

for drawing a value uniformly on the interval \([0, 1]\). The second type class defines a single function

\[
\text{class (MonadSample } m, \text{ MonadCond } m) \Rightarrow \text{MonadInfer } m
\]

for scoring the execution path. The combined typeclass

\(\text{Commit 59a50b1.}\)

\(\text{https://github.com/adscib/monad-bayes}\)
is simply for convenience. Models in MonadBayes have the type

```haskell
infer :: (MonadInfer m, MonadSample n) => m a
```

for a model returning a value of type `a`. These are not sampleable (if they were, we would already have the posterior). An inference method

```haskell
newtype Weighted m a = Weighted (StateT (Log Double) m a)
```

will transform the model into a sampleable monad where the result depends on the method, but is normally a list of values and their weight. The library treats inference methods as transformations of distributions that strips them of conditionals (scoring). The type constraint allows choice in interpreting the model and result, allowing for flexibility of different inference methods.

There are only three non-transformer instances of `MonadSample`: `SamplerIO`, `SamplerST` and `Enumerator`. The first sources its randomness from the world and the second from its implicit state. The third monad only allows discrete distributions. The library defines several monad transformers having instances of `MonadSample` and `MonadCond` (possibly depending on the inner monad). A monad

```haskell
instance Monad m => MonadCond (Weighted m) where
  score w = Weighted (modify (* w))
```

for accumulating likelihood with the instance

```haskell
newtype Population m a = Population (Weighted (ListT m) a)
```

simply multiplying the observed scores. A more complicated transformer

```haskell
newtype Sequential m a = Sequential (Coroutine (Await ()) m a)
```

is used for a population of particles. The `ListT` transformer adds non-determinism for running the program several times independently and `Weighted` provides sampling and scoring.

For sequential methods a transformer

```haskell
type CSMC m a = Coroutine (Yield (Log Double)) m a
```

based on coroutines is provided. It is made an instance of `MonadCond` by scoring in the underlying monad and then suspending. The inner monad may then be transformed (by for example resampling) before resuming.

### 2.3. iPMCMC

![Figure 4: Structure of the iPMCMC algorithm with $M$ nodes, $N$ particles per node, $R$ MCMC iterations and $P = M/2$.](image)

The interacting Particle Markov chain Monte Carlo sampler uses $M$ nodes indexed $m = 1, \ldots, M$ of which $P \leq M$ nodes use CSMC and the rest $(M - P)$ nodes use SMC (see Figure 4). Each node uses $N$ particles. By running both exploratory (SMC) and exploitative (CSMC) nodes a balance is possibly struck improving mixing and sample quality. Also, by running several methods concurrently there is the possibility of parallelisation.

Each node $m$ returns an estimate of
the marginal likelihood
\[ \hat{Z}_m = \prod_{t=1}^{T} \frac{1}{N} \sum_{i=1}^{N} w_{t,m}^i \] (4)
and its internal trajectories
\[ t_m = \{(x_{t,m}^i, w_{t,m}^i)\}_{i=1}^{N}. \] (5)

The \( P \) samples kept for each iteration is sampled from the nodes weighted by marginal likelihood. For each conditional node, a node is sampled from the union of all unconditional nodes and current conditional node. If the result is from an unconditional node, the current node is “switched out” (considered unconditional) and the target “switched in”. The switched-out node is considered unconditional immediately when resampling the next conditional node. The switched-in node is not considered again.

From each node \( m \) chosen this way (the \( P \) new conditional nodes) one trajectory \( t_{m}^{b} \) is chosen from its particles weighted on final particle weight \( w_{T,m}^i \) such that
\[ \Pr(b_m = i) = \frac{w_{T,m}^i}{\sum_{k} w_{T,m}^k}. \] (6)

The \( P \) samples generated on iteration \( r \) is denoted \( x'[r] \) and are used as conditional trajectories in the next MCMC iteration. A function \( f(x) \) may be estimated by
\[ \mathbb{E}[f(x)] = \frac{1}{RP} \sum_{r=1}^{R} \sum_{j=1}^{P} f(x'_j[r]) \] (7)
for total MCMC iterations \( R \). See the paper by Rainforth et al. [11] for details on the algorithm.

3. Method

The Monad-Bayes framework contains many components to design models and inference engines. In particular it already contains an SMC sampler. However, the sampler does not support providing either the estimated marginal likelihood \( \hat{Z} \) or internal trajectories required to condition the CSMC sampler. Therefore a custom SMC sampler is required.

Using resampling requires the ability to resume a computation. One solution is controlling the randomness and then continuing with a specific randomness. If the random variables are assigned the same values then the result will be the same. This requires significant bookkeeping and needs a custom sampler to instance the MonadSample type class. Another solution would be to use coroutines. Here the calculation may be suspended arbitrarily and resumed at will, simplifying the implementation. This is what the Sequential monad uses.

The current SMC sampler is based on the Sequential monad and performs a transformation on the inner monad after each step to implement resampling. The custom SMC sampler would have a complicated inner monad to track both the marginal likelihood and all trajectories. To avoid a complex and potentially expensive transformation after each step the state is instead kept separate and the Coroutine monad is used directly to control the execution. By using the MonadSample instance of the base monad and yielding the score at each suspension we also have a MonadCond instance.

To simplify the implementation, the models are required to have the same number of scorings in each execution path. This is equivalent to the columns of Figure 3 always being the same length, i.e. terminating at the same time. The resampling is then greatly simplified by not having to consider the edge case where some particles are already finished.

In summary we represent the model as a coroutine that is advanced to the next scoring, yielding the score \( w \). The score, which is used for resampling and updating the marginal likelihood \( \hat{Z} \), is kept externally. Finally the trajectories and marginal likelihood are returned.
3.1. SMC and CSMC implementation

The implementation may be found on GitHub. To simplify the types, several type synonyms are introduced. To represent the model

```haskell
type Trace m a = ( Either (CSMC m a) a , Log Double)
```

is used to add suspension via the Coroutine monad. After each scoring the coroutine suspends and yields the score \( w \). The monad \( m \) will normally be SampleIO. A Trace, defined as

```haskell
type Trace m a = ( Either (CSMC m a) a , Log Double)
```

is a snapshot of the program execution, paused at a scoring. It contains the rest of the program (or its value) and the accumulated score up to this point. The traces are collected into a list

```haskell
type SMCState m a = ( Vector (Trajectory m a) , Log Double)
```

representing a particle trajectory. The internal state of the CSMC function

```haskell
type SMCState m a = ( Vector (Trajectory m a) , Log Double)
```

hold an array of its particle trajectories and the estimated marginal likelihood.

The signature of the CSMC function is

```haskell
CsmcHelper :: MonadSample m
 => Maybe (Trajectory m a) , SMCState m a
 -> m (SMCState m a)
```

where in addition to the model the number of particles \( N \) and the conditional trajectory are supplied. To simplify the implementation, only SMCStates where either all or no trajectories simultaneously are finished are considered valid. This means the supplied model must have the same number of scorings in every execution path. This constraint is not impossible to remove but was introduced to simplify the implementation. The function \texttt{csmc} simply initiates the state and calls a helper function

```haskell
step :: MonadSample m
 => CSMC m a
 -> m (Trace m a)
```

common for both CSMC and SMC. The first argument represents the conditional trajectory split on the current suspension (executed part on the left and unexecuted part to the right), initially one empty. It is Nothing for SMC denoting no conditional trajectory. The second argument is the carried state, initially only the model with weight and marginal likelihood 1. To advance, a \texttt{stepPop} function is repeatedly applied until the model is finished (end of program).

To step the population of particles with the model a separate function

```haskell
stepPop :: MonadSample m
 => Maybe (Trajectory m a) , SMCState m a
 -> m (SMCState m a)
```

advances the model of a single particle, returning the continuation of the program and the score. It is used in the function

```haskell
type IPMCMCResult a = [V.Vector a]
```

taking the optional trajectory, the current state and returns the new state. Each particle is advanced once and then multinomial resampling is performed. In addition, the marginal likelihood \( Z_m \) is updated by multiplying with the mean of the latest scores according to Equation 4. The resampling is conditioned on the given trajectory if it exists.
3.2. iPCMCMC implementation

Again, type synomyes are used to simplify the signatures. The result of `ipmcmc`

```haskell
data IPMCMCState m a = IPMCMCState
  { numnodes :: Int -- M
  , numcond :: Int -- P
  , nummcmc :: Int -- R
  , conds :: V.Vector (Trajectory m a)
  , result :: IPMCMCResult a
  , smcnode :: m (SMCState m a)
  , csmcnode :: Trajectory m a -> m (SMCState m a)
  }
```

is a list of the values of the conditional trajectories for each iteration. It is isomorphic to a $R \times P$ matrix. The inner state of the algorithm

```haskell
ipmcmc :: (MonadFork m, MonadSample m) => Int -- N -> Int -- M -> Int -- R -> CSMC m a -> m (IPMCMCResult a)
```

contains the total number of nodes $M$, the number of conditional nodes $P$, remaining MCMC iterations $R$, the chosen conditional trajectories to be used for the next iteration, the accumulated result and two functions, aliases for `smc` and `csmc` respectively using the supplied model and number of particles $N$.

At the top is the actual sampler

```haskell
ipmcmc :: (MonadFork m, MonadSample m) => Int -- N -> Int -- M -> Int -- R -> CSMC m a -> m (IPMCMCResult a)
```

taking the number of particles per node $N$, the number of nodes $M$ and number of MCMC iterations $R$ in addition to the model. The `MonadFork` constraint enables concurrent evaluation on the monad. Rainforth et al. [11] found that $P = M/2$ is the optimal number of conditional nodes, so we use the same heuristic. In the first iteration there are no conditional trajectories available, and unconditional SMC is used for all nodes in the first iteration. Using `replicateM`, `forM` and `forkExec` from the `Control.Monad.Parallel` package the nodes are evaluated in parallel.

The MCMC step takes the results of the conditional and unconditional nodes and returns the next conditional trajectories $x'[r]$. The nodes and trajectories from the nodes are sampled according to section 2.3 and Equation 6 using helper functions

```haskell
dice_soft :: MonadInfer d => d Int
  dice_soft = do
    let die = uniformD [1..6]
    result <- liftM2 (+) die die
    factor (1 / fromIntegral result)
    return result
```

where `sampleNode` takes the current conditional node and the (currently) unconditional nodes, returning the sampled node and new unconditional nodes. The function `sampleCond` simply samples one trajectory from a specific node.

4. Results

The implemented method is evaluated by testing it for correctness and performance. To test the method the simple dice model

```haskell
dice_soft :: MonadInfer d => d Int
  dice_soft = do
    let die = uniformD [1..6]
    result <- liftM2 (+) die die
    factor (1 / fromIntegral result)
    return result
```
is used where two dice are added and conditioning it with a likelihood that is the reciprocal of the result. The model’s simplicity allows for calculating the exact posterior via exhaustive enumeration as well as fast testing.
The dissimilarity between the resulting and true distribution is measured by the standard Kullback-Leibler divergence metric \( D_{KL}(P \parallel Q) \) defined by
\[
D_{KL}(P \parallel Q) = -\sum_i P(i) \log \frac{Q(i)}{P(i)}
\] (8)
for the divergence from \( Q \) (the truth) to \( P \) (the result) where \( P(i) \) and \( Q(i) \) are the probability of the samples. For a correct method the error should decay according to a power law, giving a straight line in a log-log plot.

4.1. Parallelisation
First the parallelisation is examined. The iPMCMC method is run on the model with \( M = 32 \), \( N = 100 \) and \( R = 10 \), and the time elapsed is measured when running on a 32-core AMD Opteron 6274 machine. For each number of threads \( 1 \leq k \leq 32 \), 10 measurements are taken and then averaged. The result is given in Figure 5a. The data should follow Amdahl’s Law [7] for speedup as the number of cores available increases (or equivalently, the number of threads available assuming available cores). It is often formulated as the speedup achieved but might easily be reformulated into
time elapsed
\[
T(k) = T(1) \left( 1 - \frac{1}{k} P \right)
\] (9)
for a given parallelisation \( P \), which we want to find. The function is not linear in \( k \), but is linear in \( \frac{1}{k} \). This allows us to use linear least-squares regression to find \( P \) if we plot the time elapsed against the reciprocal of the number of threads available, see Figure 5b. Both the slope \( a \) and intercept \( b \) may be used to calculate \( P \), so the average of each method is used, giving approximately \( P \approx 0.86 \).

4.2. Correctness
Correctness may be demonstrated by comparing the results obtained from the method with the exact posterior distribution. For evaluation we choose \( M = 32 \), \( N = 100 \) and \( R = 1000 \) where \( M \) and \( N \) are the same as the evaluation in the original article [11] and \( R \) is chosen to keep the testing time reasonable. For parallelisation, a value \( k = 20 \) was chosen, motivated by the previous result. The sampler returns a list of samples for each iteration, allowing calculation of the error for any number of iterations \( 1 \leq r \leq R \) by taking the first \( r \) elements of the result. For each \( r \) the KL divergence is calculated according to Equation 8. To stabilise the results, the sampler is run 10 times and for each \( r \) a 95% confidence interval for the error is obtained. The result is given in Figure 6. For detailed runtime information, see appendix A.

The iPMCMC method implemented in this thesis is given by “iPMCMC (original)”. It does not converge to the true posterior as its not decreasing linearly but instead settles on some large error. It uses 33 MB of memory for a single run.

A variant “iPMCMC (variant)” that uses simple uniform sampling when selecting the nodes instead of their marginal likelihood as described in section 2.3 shows the correct behaviour. It uses 659 MB of memory for a single run.

For reference, a single SMC sampler (the existing SMC sampler in Monad-Bayes) with increasing number of particles, “SMC (multinomial)” (12 MB for \( N = 4096 \)), and a trivially parallelisable alternative using no conditional samplers and uniformly sampling \( P \) particles at each iteration (3121 MB), are also given.

4.3. Evaluation
The parallelisation described in section 4.1 is promising. The result indicated that over 80% of the algorithm as implemented is parallelisable and the result in section 4.2 shows that the variant algorithm outperforms the single SMC sampler after a few seconds.

The result of the original iPMCMC implementation is more troubling. The existence of
5. Conclusion

This thesis shows that the approach by aggressively parallelising simpler samplers for use in MCMC methods is fruitful, evident by the performance comparison in Figure 6 and the parallelisation analysis in section 4.1. However, the implementation of such methods are prone to error. The coordination of the many samplers is sensitive and great care have to be taken to not introduce bugs when implementing a procedure in functional code.

5.1. Limitations

The iPMMCMC method implemented in this thesis is simplified. As written it only supports a subset of the models expressible in Monad-Bayes. To simplify the resampling, only models containing the same number of scorings in every possible execution path are supported. In addition, the article by Rainforth et al. describes an approach for using all $MN$ generated trajectories each iteration instead of just $P$ trajectories. Finally, the implementation could be made more performant. The pure nature of Haskell results in much copying of data, but batched updates may be sped up in a safe way by performing mutation in the ST monad [9].

The evaluation only compares the implementation with samplers found in Monad-
Bayes, in particular a comparison to other MCMC methods is missing. Another interesting comparison would be comparing the implementation presented here to the iPMCMC implementation in Anglican. Another aspect of the tests is the simplicity of the model. The model was chosen both because it has an exact calculable posterior but also because it is fast. More complicated models, for example hidden Markov models, have calculable posteriors.

5.2. Future

The points brought up on the previous section could be addressed in the future. More time could be spent on the implementation to eliminate the bugs and artificial limitations and truly evaluate the iPMCMC method in Monad-Bayes.

The evaluation could be made more complete by comparing to additional methods and other probabilistic programming languages. More models could also be tested to see if the trends apparent for the simple model are true in more complex models.

References


A. Runtime information

The runtime information reported by GHC using `+RTS -N20 -s`.

A.1. iPMCMC

Runtime information for the method implemented in this thesis using $M = 32$, $N = 100$ and $R = 1000$.

A.1.1. Original

The method implemented to the specification by Rainforth et al.

35,951,853,136 bytes allocated in the heap
6,632,103,336 bytes copied during GC
7,618,296 bytes maximum residency (776 sample(s))
978,784 bytes maximum slop
33 MB total memory in use (0 MB lost due to fragmentation)

<table>
<thead>
<tr>
<th>Tot time (elapsed)</th>
<th>Avg pause</th>
<th>Max pause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gen 0 11125 colls, 11125 par</td>
<td>583.515s 89.322s 0.0080s 0.0520s</td>
<td></td>
</tr>
<tr>
<td>Gen 1 776 colls, 775 par</td>
<td>83.877s 9.302s 0.0120s 0.0491s</td>
<td></td>
</tr>
</tbody>
</table>

Parallel GC work balance: 34.77% (serial 0%, perfect 100%)

TASKS: 42 (1 bound, 41 peak workers (41 total), using -N20)

SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

| INIT time | 0.001s (0.012s elapsed) |
| MUT time | 1917.900s (196.744s elapsed) |
| GC time | 667.393s (98.624s elapsed) |
| EXIT time | 0.003s (0.003s elapsed) |
| Total time | 2585.459s (295.384s elapsed) |

Alloc rate 18,745,422 bytes per MUT second

Productivity 74.2% of total user, 66.6% of total elapsed

gc_alloc_block_sync: 2114508
whitehole_spin: 0
gen[0].sync: 72407
gen[1].sync: 122141

A.1.2. Uniform node sampling

Like the original, but sampling the nodes (for conditional trajectories) uniformly instead of by marginal likelihood $\hat{Z}_m$. 

17
35,839,146,096 bytes allocated in the heap
7,586,027,520 bytes copied during GC
233,237,360 bytes maximum residency (64 sample(s))
7,742,744 bytes maximum slop
  659 MB total memory in use (0 MB lost due to fragmentation)

<table>
<thead>
<tr>
<th></th>
<th>Tot time (elapsed)</th>
<th>Avg pause</th>
<th>Max pause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gen 0</td>
<td>11342 colls, 11342 par</td>
<td>641.414s</td>
<td>95.252s</td>
</tr>
<tr>
<td>Gen 1</td>
<td>64 colls, 63 par</td>
<td>38.838s</td>
<td>3.696s</td>
</tr>
</tbody>
</table>

Parallel GC work balance: 37.32% (serial 0%, perfect 100%)

TASKS: 42 (1 bound, 41 peak workers (41 total), using -N20)

SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

INIT time 0.003s ( 0.012s elapsed)
MUT time 1934.201s (194.125s elapsed)
GC time 680.253s ( 98.948s elapsed)
EXIT time 0.015s ( 0.138s elapsed)

Total time 2614.627s (293.223s elapsed)

Alloc rate 18,529,173 bytes per MUT second

Productivity 74.0% of total user, 66.3% of total elapsed

gc_alloc_block_sync: 8771449
whitehole_spin: 0
gen[0].sync: 71306
gen[1].sync: 611021

A.1.3. Uniform trajectory sampling

Like the original, but sampling the trajectories (for conditional trajectories) uniformly instead of by final trajectory score \(w_T\).

35,664,262,816 bytes allocated in the heap
6,741,223,040 bytes copied during GC
7,704,880 bytes maximum residency (781 sample(s))
776,656 bytes maximum slop
  32 MB total memory in use (0 MB lost due to fragmentation)

<table>
<thead>
<tr>
<th></th>
<th>Tot time (elapsed)</th>
<th>Avg pause</th>
<th>Max pause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gen 0</td>
<td>10521 colls, 10521 par</td>
<td>575.589s</td>
<td>86.493s</td>
</tr>
<tr>
<td>Gen 1</td>
<td>781 colls, 780 par</td>
<td>83.529s</td>
<td>9.280s</td>
</tr>
</tbody>
</table>

Parallel GC work balance: 33.38% (serial 0%, perfect 100%)

TASKS: 42 (1 bound, 41 peak workers (41 total), using -N20)

SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

INIT time 0.002s ( 0.012s elapsed)
MUT time 1917.316s (194.941s elapsed)
GC time 659.119s (95.773s elapsed)
EXIT time 0.000s (0.009s elapsed)
Total time 2576.590s (290.735s elapsed)

Alloc rate 18,601,144 bytes per MUT second

Productivity 74.4% of total user, 67.1% of total elapsed

gc_alloc_block_sync: 2333836
whitehole_spin: 0
gen[0].sync: 91490
gen[1].sync: 112733

A.1.4. Uniform node and trajectory sampling

Like the original, but sampling the trajectories and nodes uniformly.

35,450,584,888 bytes allocated in the heap
7,706,953,000 bytes copied during GC
221,721,096 bytes maximum residency (74 sample(s))
8,304,336 bytes maximum slop
627 MB total memory in use (0 MB lost due to fragmentation)

<table>
<thead>
<tr>
<th></th>
<th>Tot time (elapsed)</th>
<th>Avg pause</th>
<th>Max pause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gen 0</td>
<td>10793 colls, 10793 par</td>
<td>593.880s</td>
<td>90.989s</td>
</tr>
<tr>
<td>Gen 1</td>
<td>74 colls, 73 par</td>
<td>44.216s</td>
<td>4.081s</td>
</tr>
</tbody>
</table>

Parallel GC work balance: 36.17% (serial 0%, perfect 100%)

TASKS: 42 (1 bound, 41 peak workers (41 total), using -N20)

SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC’d, 0 fizzled)

INIT time 0.004s (0.010s elapsed)
MUT time 1903.107s (192.618s elapsed)
GC time 638.096s (95.070s elapsed)
EXIT time 0.002s (0.004s elapsed)
Total time 2541.367s (287.702s elapsed)

Alloc rate 18,627,743 bytes per MUT second

Productivity 74.9% of total user, 67.0% of total elapsed

gc_alloc_block_sync: 8563158
whitehole_spin: 0
gen[0].sync: 64290
gen[1].sync: 676784

A.2. Many SMC

The trivial parallelisation of M SMC samplers taking P particles each iteration with M = 32, N = 100 and \( R = 1000. \)
63,750,517,704 bytes allocated in the heap
12,467,705,232 bytes copied during GC
1,288,921,864 bytes maximum residency (24 sample(s))
22,955,776 bytes maximum slop
3121 MB total memory in use (0 MB lost due to fragmentation)

 Tot time (elapsed)  Avg pause  Max pause
Gen 0  39464 colls, 39464 par  1744.526s  243.761s  0.0062s  0.0514s
Gen 1  24 colls,  23 par  95.717s  8.669s  0.3612s  2.0314s

Parallel GC work balance: 24.15% (serial 0%, perfect 100%)

TASKS: 43 (1 bound, 42 peak workers (42 total), using -N20)

SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

INIT time 0.005s ( 0.013s elapsed)
MUT time 2572.475s (251.723s elapsed)
GC time 1840.243s (252.431s elapsed)
EXIT time 0.150s ( 0.745s elapsed)

Total time 4413.036s (504.911s elapsed)
Alloc rate 24,781,783 bytes per MUT second
Productivity 58.3% of total user, 50.0% of total elapsed

gc_alloc_block_sync: 20518554
whitehole_spin: 0
gen[0].sync: 35326
gen[1].sync: 1123794

A.3. SMC
A single SMC sampler with $N = 4096$.

730,759,592 bytes allocated in the heap
37,668,168 bytes copied during GC
4,327,112 bytes maximum residency (8 sample(s))
116,352 bytes maximum slop
12 MB total memory in use (0 MB lost due to fragmentation)

 Tot time (elapsed)  Avg pause  Max pause
Gen 0  1390 colls,  0 par  0.135s  0.126s  0.0001s  0.0014s
Gen 1   8 colls,  0 par  0.058s  0.066s  0.0082s  0.0201s

TASKS: 4 (1 bound, 3 peak workers (3 total), using -N1)

SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

INIT time 0.000s ( 0.002s elapsed)
MUT time 5.981s ( 6.037s elapsed)
GC time 0.193s ( 0.191s elapsed)
EXIT time 0.002s ( 0.002s elapsed)
Total time 6.334s (6.232s elapsed)
Alloc rate 122,178,269 bytes per MUT second
Productivity 97.0% of total user, 96.9% of total elapsed

gc_alloc_block_sync: 0
whitehole_spin: 0
gen[0].sync: 0
gen[1].sync: 0