A Compilation Target for Probabilistic Programming Languages

Brooks Paige   BROOKS@ROBOTS.OX.AC.UK
Frank Wood   FWOOD@ROBOTS.OX.AC.UK
University of Oxford, Department of Engineering Science, Oxford, UK

Abstract

Forward inference techniques such as sequential Monte Carlo and particle Markov chain Monte Carlo for probabilistic programming can be implemented in any programming language by creative use of standardized operating system functionality including processes, forking, mutexes, and shared memory. Exploiting this we have defined, developed, and tested a probabilistic programming language intermediate representation language we call probabilistic C, which itself can be compiled to machine code by standard compilers and linked to operating system libraries yielding an efficient, scalable, portable probabilistic programming compilation target. This opens up a new hardware and systems research path for optimizing probabilistic programming systems.

1. Introduction

Compilation is source to source transformation. We use the phrase intermediate representation to refer to a target language for a compiler. This paper introduces a C-language library that makes possible a C-language intermediate representation for probabilistic programming languages that can itself be compiled to executable machine code. We call this intermediate language *probabilistic C*. Probabilistic C can be compiled normally and uses only macros and `posix` operating system libraries (Open Group, 2004a) to implement general-purpose, scalable, parallel probabilistic programming inference. Note that in this paper we do not show how to compile any existing probabilistic programming languages (i.e. IBAL (Pfeffer, 2001), BLOG (Milch et al., 2007), Church (Goodman et al., 2008), Figaro (Pfeffer, 2009), Venture (Mansinghka et al., 2013), or Anglican (Wood et al., 2014)) to this intermediate representation; instead we leave this to future work noting that there is a wealth of readily available resources on language to language compilation that could be leveraged to do this. We instead characterize the performance of the intermediate representation itself by writing programs directly in probabilistic C and then testing them on computer architectures and programs that illustrate the capacities and trade-offs of both the forward inference strategies probabilistic C employs and the operating system functionality on which it depends.

Probabilistic C programs compile to machine executable meta-programs that perform inference over the original program via forward methods such as sequential Monte Carlo (Doucet et al., 2001) and particle MCMC variants (Andrieu et al., 2010). Such inference methods can be implemented in a sufficiently general way so as to support inference over the space of probabilistic program execution traces using just `posix` operating system primitives.

1.1. Related work

The characterization of probabilistic programming inference we consider here is the process of sampling from the a posteriori distribution of execution traces arising from stochastic programs constrained to reflect observed data. This is the view taken by the Church (Goodman et al., 2008), Venture (Mansinghka et al., 2013), and Anglican (Wood et al., 2014) programming languages among others. In such languages, models for observed data can be described purely in terms of a forward generative process.

Markov chain Monte Carlo (MCMC) is used by these systems to sample from the posterior distribution of program execution traces. Single-site Metropolis Hastings (MH) (Goodman et al., 2008) and particle MCMC (PMCMC) (Wood et al., 2014) are two such approaches. In the latter it was noted that a `fork`-like operation is a fundamental requirement of forward inference methods for probabilistic programming, where `fork` is the standard `posix` operating system primitive (Open Group, 2004b). Kiselyov & Shan (2009) also noted that delimited continuations, a user-level generalization of `fork` could be used for inference, albeit in a restricted family of models.

2. Probabilistic Programming

Any program that makes a random choice over the course of its execution implicitly defines a prior distribution over its random variables; running the program can be interpreted as drawing a sample from the prior. Inference in

```
#include "probabilistic.h"

int main(int argc, char **argv) {

    double var = 2;
    double mu = normal_rng(1, 5);

    observe(normal_lnp(9, mu, var));
    observe(normal_lnp(8, mu, var));

    predictf("mu,%f\n", mu);

    return 0;
}
```

Figure 1. Two data points, distributed according to a Gaussian with unknown mean μ. We place a $\mathcal{N}(1,5)$ prior on μ, and observe two data points drawn from $\mathcal{N}(\mu,2)$.

probabilistic programs involves conditioning on observed data, and characterizing the posterior distribution of the random variables given data. We introduce probabilistic programming capabilities into C by providing a library with two functions: `observe` which conditions the program execution trace given the log-likelihood of a data point, and `predict` which marks expressions for which we want posterior samples. Any random number generator and sampling library can be used for making random choices in the program, any numeric log likelihood value can be passed to an `observe`, and any C expression which can be printed can be reported using `predict`. The library includes a single macro which renames `main` and wraps it in another function that runs the original in an inner loop in the forward inference algorithms to be described.

Figure 1 shows a simple probabilistic C program for estimating the posterior distribution for the mean of a Gaussian, conditioned on two observed data points y_1, y_2, corresponding to the model

$$\mu \sim \mathcal{N}(1,5), \qquad y_1, y_2 \stackrel{iid}{\sim} \mathcal{N}(\mu,2). \qquad (1)$$

We `observe` the data y_1, y_2 and `predict` the posterior distribution of μ. The functions `normal_rng` and `normal_lnp` in Figure 1 return (respectively) a normally-distributed random variate and the log probability density of a particular value, with mean and variance parameters `mu` and `var`. The `observe` statement requires only the log-probability of the data points 8 and 9 conditioned on the current program state; no other information about the likelihood function or the generative process. In this program we predict the posterior distribution of a single value `mu`.

Although C is a comparatively low-level language, it can nonetheless represent many well-known generative models concisely and transparently. A hidden Markov model example is shown in Figure 2, in which N observed data points $y_{1:N}$ are drawn from an underlying Markov chain with K latent states, each with Gaussian emission distributions with mean μ_k, and a (known) $K \times K$ state transition

```
#include "probabilistic.h"
#define K 3
#define N 11

/* Markov transition matrix */
static double T[K][K] = { { 0.1,  0.5,  0.4 },
                          { 0.2,  0.2,  0.6 },
                          { 0.15, 0.15, 0.7 } };

/* Observed data */
static double data[N] = { NAN, .9, .8,  .7,   0, -.025,
                               -5, -2, -.1,   0,  0.13 };

/* Prior distribution on initial state */
static double initial_state[K] = { 1.0/3, 1.0/3, 1.0/3 };

/* Per-state mean of Gaussian emission distribution */
static double state_mean[K] = { -1, 1, 0 };

/* Generative program for a HMM */
int main(int argc, char **argv) {

    int states[N];
    for (int n=0; n<N; i++) {
        states[n] = (n==0) ? discrete_rng(initial_state, K)
                           : discrete_rng(T[states[n-1]], K);
        if (n > 0) {
            observe(normal_lnp(data[n], state_mean[states[n]], 1));
        }
        predictf("state[%d],%d\n", n, states[n]);
    }

    return 0;
}
```

Figure 2. A hidden Markov model (HMM) with 3 underlying states and Gaussian emissions, observed at 10 discrete time-points. We observe 10 data points and predict the marginal distribution over latent `state` at each time point. The `predictf` directive is a `predict` which formats output using standard `printf` semantics.

matrix T, such that

$$z_0 \sim \text{Discrete}([1/K, \ldots, 1/K]) \qquad (2)$$
$$z_n | z_{n-1} \sim \text{Discrete}(T_{z_{n-1}}) \qquad (3)$$
$$y_n | z_n \sim \mathcal{N}(\mu_{z_n}, \sigma^2). \qquad (4)$$

Bayesian nonparametric models can also be represented concisely; in Figure 3 we show a generative model for an infinite mixture of Gaussians, using a Chinese restaurant process (CRP) to sample non-negative integer partition assignments ϕ_n for each data point $y_1, \ldots, y_N$. For each partition, mean and variance parameters $\mu_{\phi_n}, \sigma^2_{\phi_n}$ are drawn from a Normal-Gamma prior; the data points y_n themselves are drawn from a normal distribution with parameters ϕ_n, with the full generative model

$$\phi_n \sim \text{CRP}(\alpha) \qquad (5)$$
$$1/\sigma^2_{\phi_n} \sim \text{Gamma}(1,1) \qquad (6)$$
$$\mu_{\phi_n} | \sigma^2_{\phi_n} \sim \mathcal{N}(0, \sigma^2_{\phi_n}) \qquad (7)$$
$$y_n | \phi_n, \mu_{\phi_n}, \sigma^2_{\phi_n} \sim \mathcal{N}(\mu_{\phi_n}, \sigma^2_{\phi_n}). \qquad (8)$$

2.1. Operating system primitives

Inference proceeds by drawing posterior samples from the space of program execution traces. We define an execution

```c
#include "probabilistic.h"
#define N 10

// Observed data
static double data[N] = { 1.0,  1.1,   1.2,
                         -1.0, -1.5,  -2.0,
                          0.001, 0.01, 0.005, 0.0 };

// Struct holding mean and variance for each cluster
typedef struct theta {
    double mu;
    double var;
} theta;

// Draws a sample of theta from a normal-gamma prior
theta draw_theta() {
    double variance = 1.0 / gamma_rng(1, 1);
    return (theta) { normal_rng(0, variance), variance };
}

int main(int argc, char **argv) {

    polya_urn_state urn;
    double alpha = 1.0;
    polya_urn_new(&urn, alpha);

    theta params[N];
    bool known_theta[N] = { false };

    for (int n=0; n<N; n++) {
        int class = polya_urn_draw(&urn);
        if (!known_theta[class]) {
            params[class] = draw_theta();
            known_theta[class] = true;
        }
        observe(normal_lnp(data[n], params[class].mu,
                                    params[class].var));
    }

    // Predict number of classes
    predictf("num_classes,%2d\n", urn.len_buckets);

    // Release memory; exit
    polya_urn_free(&urn);
    return 0;
}
```

Figure 3. A infinite mixture of Gaussians on the real line. Class id variables for each of the $N = 10$ data points are drawn, following a Blackwell-MacQueen urn scheme to sequentially sample from a Dirichlet process; for each class id, parameters θ of a normal likelihood function are drawn from a normal-gamma prior. The urn scheme of polya_urn_draw and its associated data structure polya_urn_state are implemented in a separate file.

trace as the sequence of memory states (stack frames and allocated memory contents) that arises during the sequential step execution of machine instructions.

The algorithms we propose for inference in probabilistic programs map directly onto standard computer operating system constructs, exposed in POSIX-compliant operating systems including Linux, BSD, and Mac OS X. The cornerstone of our approach is POSIX fork (Open Group, 2004b). When a process forks, it clones itself, creating a new process with an identical copy of the execution state of the original process, and identical source code; both processes then continue with normal program execution completely independently from the point where fork was called. While copying program execution state may naïvely sound like a costly operation, this actually can be rather efficient: when fork is called, a lazy copy-on-write procedure is used to avoid deep copying the entire program

memory. Instead, initially only the pagetable is copied to the new process; when an existing variable is modified in the new program copy, then and only then are memory contents duplicated. The overall cost of forking a program is proportional to the fraction of memory which is rewritten by the child process (Smith & Maguire, 1988).

Using fork we can branch a single program execution state and explore many possible downstream execution paths. Each of these paths runs as its own process, and will run in parallel with other processes. In general, multiple processes run in their own memory address space, and do not communicate or share state. We handle inter-process communication via a small shared memory segment; the details of what global data must be stored are provided later.

Synchronization between processes is handled via mutual exclusion locks (*mutex* objects). Mutexes become particularly useful for us when used in conjunction with a synchronized counter to create a *barrier*, a high-level blocking construct which prevents any process proceeding in execution state beyond the barrier until some fixed number of processes have arrived.

3. Inference

3.1. Probability of a program execution trace

To notate the probability of a program execution trace, we enumerate all N observe statements, and the associated observed data points $y_1, \ldots, y_N$. During a single run of the program, some total number N' random choices $\mathbf{x}'_1, \ldots, \mathbf{x}'_{N'}$ are made. While N' may vary between individual executions of the program, we require is that N is constant. That is, valid probabilistic models correspond to probabilistic C program with the same number of observe directive calls on every program execution. As it is left to the programmer (or potentially the compiler; see Section 5 for a brief discussion of this point) it is quite easy to write models in probabilistic C that are statistically difficult to reason about. This is the primary reason for proposing probabilistic C as a compilation target. In many probabilistic programming languages, for instance Anglican (Wood et al., 2014) and Venture (Mansinghka et al., 2013), such constraints are explicitly imposed by the language itself.

The observations y_n can appear at any point in the program source code and define a partition of the random choices $\mathbf{x}'_{1:N'}$ into N subsequences $\mathbf{x}_{1:N}$, where each $\mathbf{x}_n$ contains all random choices made up to observing y_n but excluding any random choices prior to observation y_{n-1}. We can then define the probability of any single program execution trace as

$$p(y_{1:N}, \mathbf{x}_{1:N}) = \prod_{n=1}^{N} g(y_n | \mathbf{x}_{1:n}) f(\mathbf{x}_n | \mathbf{x}_{1:n-1}) \quad (9)$$

In this manner, any model with a generative process that can be described in arbitrary C code can be represented in this sequential form in the space of program execution traces.

Each `observe` statement takes as its argument $\ln g(y_n|\mathbf{x}_{1:n})$. Each quantity of interest in a `predict` statement corresponds to some deterministic function $h(\cdot)$ of all random choices $\mathbf{x}_{1:N}$ made during the execution of the program. Given a set of S posterior samples $\{\mathbf{x}_{1:N}^{(s)}\}$, we can approximate the posterior distribution of the `predict` value as

$$h(\mathbf{x}_{1:N}) \approx \frac{1}{S}\sum_{s=1}^{S} h(\mathbf{x}_{1:N}^{(s)}). \qquad (10)$$

3.2. Sequential Monte Carlo

Forward simulation-based algorithms are a natural fit for probabilistic programs: run the program and report executions that match the data. Sequential Monte Carlo (SMC, sequential importance resampling) forms the basic building block of other, more complex particle-based methods, and can itself be used as a simple approach to probabilistic programming inference. SMC approximates a target density $p(\mathbf{x}_{1:N}|y_{1:N})$ as a weighted set of L realized trajectories $\mathbf{x}_{1:N}^{\ell}$ such that

$$p(\mathbf{x}_{1:N}|y_{1:N}) \approx \sum_{\ell=1}^{L} w_N^{\ell} \delta_{\mathbf{x}_{1:N}^{\ell}}(\mathbf{x}_{1:N}). \qquad (11)$$

For most probabilistic programs of interest, it will be intractable to sample from $p(\mathbf{x}_{1:N}|y_{1:N})$ directly. Instead, noting that (for $n > 1$) we have the recursive identity

$$p(\mathbf{x}_{1:n}|y_{1:n}) \qquad (12)$$
$$= p(\mathbf{x}_{1:n-1}|y_{1:n-1})g(y_n|\mathbf{x}_{1:n})f(\mathbf{x}_n|\mathbf{x}_{1:n-1}),$$

we sample from $p(\mathbf{x}_{1:N}|y_{1:N})$ by iteratively sampling from each $p(\mathbf{x}_{1:n}|y_{1:n})$, in turn, from 1 through N. At each n, we construct an importance sampling distribution by proposing from some distribution $q(\mathbf{x}_n|\mathbf{x}_{1:n-1}, y_{1:n})$; in probabilistic programs we find it convenient to propose directly from the executions of the program, i.e. each sequence of random variates $\mathbf{x}_n$ is jointly sampled from the program execution state dynamics

$$\mathbf{x}_n^{\ell} \sim f(\mathbf{x}_n|\mathbf{x}_{1:n-1}^{a_{n-1}^{\ell}}) \qquad (13)$$

where a_{n-1}^{ℓ} is an "ancestor index," the particle index $1, \dots, L$ of the parent (at time $n-1$) of $\mathbf{x}_n^{\ell}$. The unnormalized particle importance weights at each observation y_n are simply the `observe` data likelihood

$$\tilde{w}_n^{\ell} = g(y_{1:n}, \mathbf{x}_{1:n}^{\ell}) \qquad (14)$$

Algorithm 1 Parallel SMC program execution

Assume: N observations, L particles

launch L copies of the program   (parallel)
for $n = 1 \dots N$ **do**
   wait until all L reach `observe` y_n (barrier)
   update unnormalized weights $\tilde{w}_n^{1:L}$ (serial)
   if $ESS < \tau$ **then**
      sample number of offspring $O_n^{1:L}$ (serial)
      set weight $\tilde{w}_n^{1:L} = 1$ (serial)
      for $\ell = 1 \dots L$ **do**
         fork or exit   (parallel)
      end for
   else
      set all number of offspring $O_n^{\ell} = 1$ (serial)
   end if
   continue program execution   (parallel)
end for
wait until L program traces terminate   (barrier)
`predict` from L samples from $\hat{p}(\mathbf{x}_{1:N}^{1:L}|y_{1:N})$ (serial)

---

which can be normalized as

$$w_n^{\ell} = \frac{\tilde{w}_n^{\ell}}{\sum_{\ell=1}^{L} \tilde{w}_n^{\ell}}. \qquad (15)$$

After each step n, we now have a weighted set of execution traces which approximate $p(\mathbf{x}_{1:n}|y_{1:n})$. As the program continues, traces which do not correspond well with the data will have weights which become negligibly small, leading in the worst case to all weight concentrated on a single execution trace. To counteract this deficiency, we resample from our current set of L execution traces after each observation y_n, according to their weights w_n^{ℓ}. This is achieved by sampling a count O_n^{ℓ} for the number of "offspring" of a given execution trace ℓ to be included at time $n + 1$; any sampling scheme must ensure $\mathbb{E}[O_n^{\ell}] = w_n^{\ell}$. Sampling offspring counts O_n^{ℓ} is equivalent to sampling ancestor indices a_n^{ℓ}. Program execution traces with no offspring are killed; program execution traces with more than one offspring are forked multiple times. After resampling, all weights $w_n^{\ell} = 1$.

We only resample if the effective sample size

$$ESS \approx \frac{1}{\sum_{\ell}(w_n^{\ell})^2} \qquad (16)$$

is less than some threshold value τ; we choose $\tau = L/2$.

In probabilistic C, each `observe` statement forms a barrier: parallel execution cannot continue until all particles have arrived at the `observe` and have reported their current unnormalized weight. As execution traces arrive at the `observe` barrier, they take the number of particles which have already reached the current `observe` as a (temporary)

unique identifier. Program execution is then blocked as the effective sample size is computed and the number of offspring are sampled. The number of offspring are stored in a shared memory block; when the number of offspring are computed, each particle uses the identifier assigned when reaching the `observe` barrier to retrieve (asynchronously) from shared memory the number of children to `fork`. Particles with no offspring wait for any child processes to complete execution, and terminate; particles with only one offspring do not `fork` any children but continue execution as normal.

The SMC algorithm is outlined in Algorithm 1, with annotations for which steps are executed in parallel, serially, or form a barrier. After a single SMC sweep is complete, we sample values for each `predict`, and then (if desired) repeat the process, running a new independent particle filter, to draw an additional batch of samples.

3.3. Particle Metropolis-Hastings

Particle Markov chain Monte Carlo, introduced in Andrieu et al. (2010), uses sequential Monte Carlo to generate high-dimensional proposal distributions for MCMC. The most simple formulation is the particle independent Metropolis-Hastings algorithm. After running a single particle filter sweep, we compute an estimate of the marginal likelihood,

$$\hat{Z} \equiv p(y_{1:N}) \approx \prod_{n=1}^{N} \left[\frac{1}{L} \sum_{\ell=1}^{L} w_n^{\ell} \right]. \qquad (17)$$

We then run another iteration of sequential Monte Carlo which we use as a MH proposal; we estimate the marginal likelihood $\hat{Z}'$ of the new proposed particle set, and then with probability $\min(1, \hat{Z}'/\hat{Z})$ we accept the new particle set and output a new set of `predict` samples, otherwise outputting the same `predict` samples as in the previous iteration.

The inner loop of Algorithm 2 is otherwise substantially similar to SMC.

3.4. Particle Gibbs

Particle Gibbs is a particle MCMC technique that has SMC at its core as well. It has better theoretical statistical convergence properties than PIMH, but may suffer due to degeneracy concerns in some models, and requires additional computational overhead. We initialize particle Gibbs by running a single sequential Monte Carlo sweep, and then alternate between

1. sampling a single execution trace $\hat{\mathbf{x}}_{1:M}$ from the set of L weighted particles, and

2. running a "conditional" SMC sweep, in which we

Algorithm 2 Parallel PIMH program execution

Assume: M iterations, N observations, L particles

  for $m = 1 \ldots M$ **do**
    launch L copies of the program   (parallel)
    for $n = 1 \ldots N$ **do**
      wait until all L reach an `observe`   (barrier)
      update unnormalized weights $\tilde{w}_{1:L}$ (serial)
      if $ESS < \tau$ **then**
        update proposal evidence estimate $\hat{Z}'$ (serial)
        sample number of offspring $O_n^{1:L}$ (serial)
        set weight $\tilde{w}_n^{1:L} = 1$ (serial)
        for $\ell = 1 \ldots L$ **do**
          fork or exit   (parallel)
        end for
      else
        set all number of offspring $O_n^{\ell} = 1$ (serial)
      end if
      continue program execution   (parallel)
    end for
    wait until L program traces terminate   (barrier)
    accept or reject new particle set   (serial)
    `predict` from L samples from $\hat{p}(\mathbf{x}_{1:N}^{1:L}|y_{1:N})$ (serial)
    store current particle set $\mathbf{x}$ and evidence $\hat{Z}$ (serial)
    continue to next iteration   (parallel)
  end for

---

generate $L-1$ new particles in addition to the retained $\hat{\mathbf{x}}_{1:M}$.

The implementation based on operating system primitives is described in algorithms 3 and 4. The challenge here is that we must "retain" an execution trace, which we can later revisit to resume and branch arbitrarily many times. This is achieved by spawning off a "control" process at every observation point, which from then on manages the future of that particular execution state.

As before, processes arrive at an `observe` barrier, and when all particles have reached the observe we compute weights, and sample offspring counts O_n^{ℓ}. Particles with $O_n^{\ell} = 0$ terminate, but new child processes are no longer spawned right away. Instead, all remaining particles `fork` a new process whose execution path immediately diverges from the main codebase and enters the retain and branch loop in Algorithm 4. This new process takes responsibility for actually spawning the O_n^{ℓ} new children. The spawned child processes (and the original process which arrived at the `observe` barrier) wait (albeit briefly) at a new barrier marking the *end* of `observe` n, not continuing execution until all new child processes have been launched.

Program execution continues to the next `observe`, during which the retain / branch process waits until a full particle set reaches the end of the program. Once final weights $\tilde{w}_N^{1:L}$

Algorithm 3 Parallel Particle Gibbs program execution

Assume: M iterations, N observations, L particles

  for $m = 1 \ldots M$ **do**

    $L' \leftarrow L$ if $m = 1$, otherwise $L - 1$

    launch S' copies of the program   (parallel)

    for $n = 1 \ldots N$ **do**

      wait until all L' reach an `observe`   (barrier)

      compute weights for all particles   (serial)

      if $m > 1$ **then**

        signal num offspring to retained trace   (serial)

      end if

      for $\ell = 1 \ldots L'$ **do**

        spawn retain / branch process [Algo. 4] (parallel)

      end for

      wait until L particles finish branching   (barrier)

      continue program execution   (parallel)

    end for

    wait until L program traces terminate   (barrier)

    `predict` from L samples from $\hat{p}(\mathbf{x}_{1:N}^{1:L}|y_{1:N})$ (serial)

    select and signal particle to retain   (serial)

    wait until N processes are ready to branch   (barrier)

    continue to next iteration   (parallel)

  end for

Algorithm 4 Retain and Branch inner loop

Assume: input initial $C > 0$ children to spawn

  `is_retained` $\leftarrow$ `false`

  while `true` **do**

    if $C = 0$ and not `is_retained` **then**

      discard this execution trace, exit

    else $\{C \geq 0\}$

      spawn C new children

    end if

    wait for signal which resets `is_retained`

    if `is_retained` **then**

      wait for signal which resets C

    else

      discard this execution trace, exit

    end if

  end while

---

are computed, we sample (according to weight) from the final particle set to select a single particle to retain during the next SMC iteration. When the particle is selected, a signal is broadcast to all retain / branch loops indicating which process ids correspond to the retained particle; all except the retained trace exit.

The retain / branch loop now goes into waiting again (this time for a branch signal), and we begin SMC from the top of the program. As we arrive at each observe n, we only sample $L - 1$ new offspring to consider: we guarantee that at least one offspring is spawned from the retained particle at n (namely, the retained execution state at $n + 1$). However, depending on the weights, often sampling offspring will cause us to want more than a single child from the retained particle. So, we signal to the retained particle execution state at time n the number of children to spawn; the retain / branch loop returns to its entry point and resumes waiting, to see if the previously retained execution state will be retained yet again.

Note that in particle Gibbs, we must resample (select offspring and reset weights $w_n^\ell = 1$) after every observation in order to be able to properly align the retained particle on the next iteration through the program.

4. Experiments

We now turn to benchmarking probabilistic C against existing probabilistic programming engines, and evaluate the relative strengths of the three inference algorithms in Section 3. We find that compilation improves performance by 100 times or more over interpreted versions of the same inference algorithm. We also find evidence that suggests that optimising operating systems to support probabilistic programming usage could yield significant performance improvements as well.

The programs/models we use in our experiments are chosen to be sufficiently simplistic that we can compute the exact posterior distribution of interest analytically. Given the true posterior distribution p, we measure sampler performance by the KL-divergence $KL(\hat{p}||p)$, where $\hat{p}$ is our Monte Carlo estimate. The first benchmark program we consider is a hidden Markov model (HMM) very similar to that of Figure 2, where we predict the marginal distributions of each latent state. The HMM used in our experiments here is larger; it has the same model structure, but with $K = 10$ states and 50 sequential observations, and each state $k = 1, \ldots, 10$ has a Gaussian emission distribution with $\mu_k = k - 1$ and $\sigma^2 = 4$. The second benchmark is the CRP mixture of Gaussians program in Figure 3, where we predict the number of different classes.

4.1. Comparative performance of inference engines

We begin by benchmarking against two existing probabilistic programming engines: *Anglican*, as described in (Wood et al., 2014), which also implements particle Gibbs, but is an interpreted language based on Scheme, implemented in Clojure, and running on the JVM; and *probabilistic-js*[1], a compiled system implementing the inference approach in (Wingate et al., 2011), which runs Metropolis-Hastings over each individual random choice in the program execution trace. The interpreted particle Gibbs engine is mul-

---

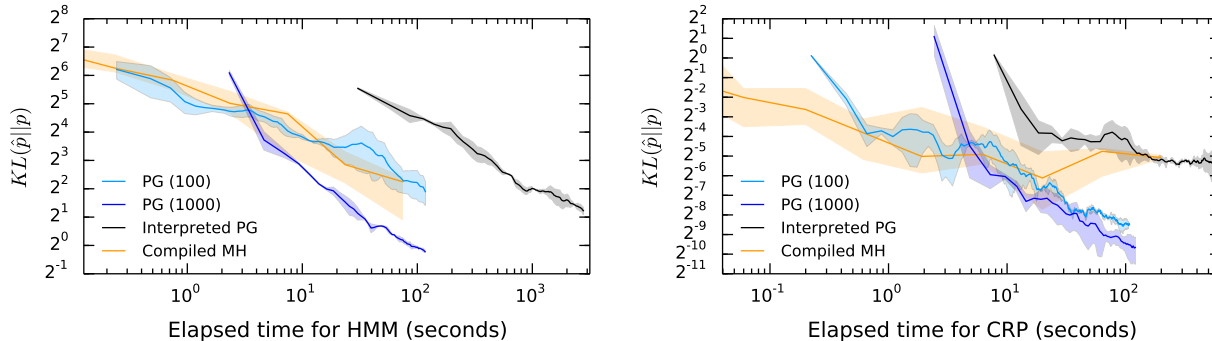[1] https://github.com/dritchie/probabilistic-js

Figure 4. Performance plots for (left) HMM and (right) CRP models, run in an 8 core computing environment on Amazon EC2. In both models we see that compiling particle Gibbs, compared to running within an interpreter, leads to a large (approx. $100\times$) constant-factor speed increase. The Metropolis-Hastings sampler converges at a similar rate as particle Gibbs with 100 particles in the HMM, but appears to mix slower asymptotically for the CRP. In both models, increasing to 1000 particles in particle Gibbs yields somewhat faster convergence at the expense of a longer waiting time until the first sample set arrives.
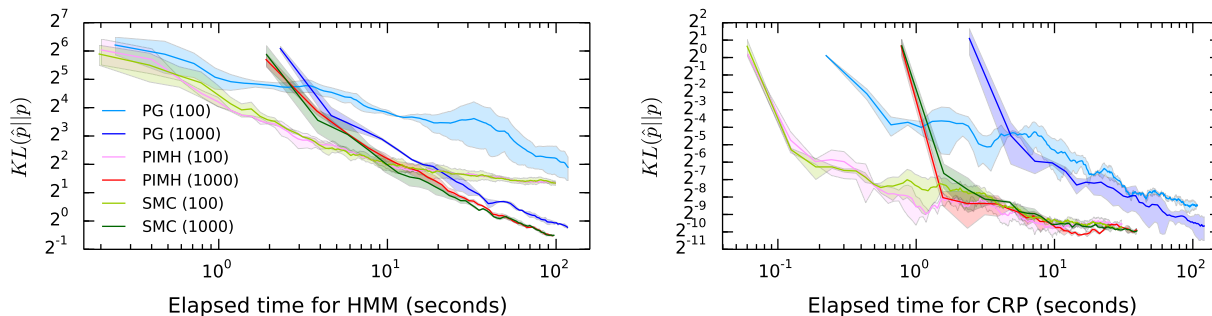

Figure 5. Comparison of SMC, PIMH, and PG for 100 and 1000 particles in (top) the CRP, and (bottom) the HMM. In the CRP, the (cheaper) SMC and PIMH algorithms converge quickly; in the HMM, both SMC and PIMH converge poorly when only 100 particles are used. Particle Gibbs also benefits from a larger particle count. Colors are the same across plots.

tithreaded, and we run it with 100 particles and 8 simultaneous threads; the Metropolis-Hastings engine only runs on a single core. In Figure 4 we compare inference performance in both of these existing engines to our particle Gibbs backend, running with 100 and 1000 particles, in an 8 core cloud computing environment on Amazon EC2, running on Intel Xeon E5-2680 v2 processors. Our compiled probabilistic C implementation of particle Gibbs runs over 100 times faster that the existing interpreted engine, generating good sample sets in on the order of tens of seconds.

The probabilistic C inference engine implements particle Gibbs, SMC, and PIMH sampling, which we compare in Figure 5 using both 100 and 1000 particles.

Figures 4 and 5 plot wall clock time against KL-divergence. We use all generated samples as our empirical posterior distribution in order to produce as fair a comparison as possible. In all engines, results are reasonably stable across runs; shaded regions denote minimum and maximum values over

multiple runs, with the median marked as a dark line. A sampler which is correctly drawing samples from the target density will show an approximately linear decrease in KL-divergence on such log-log plots. The methods based on sequential Monte Carlo do not provide any estimate of the posterior distribution until completing a single initial particle filter sweep; for large numbers of particles this may be a non-trivial amount of time. In contrast, the MH sampler begins drawing samples effectively immediately, although it may take a large number of individual steps before converging to the correct stationary distribution; individual Metropolis-Hastings samples are likely to be more autocorrelated, but producing each one is faster.

4.2. Performance characteristics across multiple cores

As the probabilistic C inference engine offloads much of the computation to underlying operating system calls, we characterize the limitations of the OS implementation by
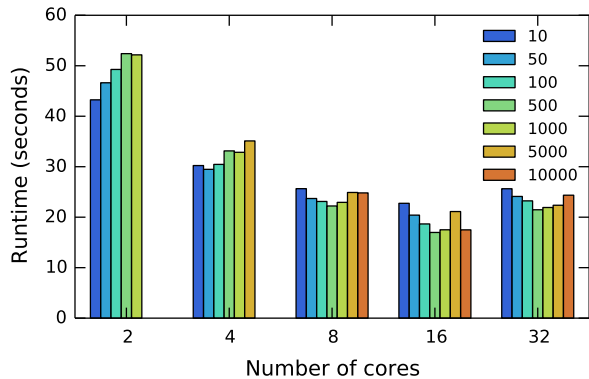
Figure 6. Effect of system architecture on runtime performance. Here we run the HMM code on EC2 instances with identical processors (horizontal axis) with varying number of particles (individual bars) and report runtime to produce 10,000 samples. Despite adding more cores, after 16 cores performance begins to degrade. Similarly, adding more particles eventually degrades performance for any fixed number of cores. In combination these suggest the availability of operating system optimizations that could improve overall performance.

comparing time to completion as we vary the number of cores. Tests for the hidden Markov model across core count (all on EC2, all with identical Intel Xeon E5-2680 v2 processors) are shown in Figure 6.

5. Discussion

Probabilistic C is a method for performing inference in probabilistic programs. Methodologically it derives from the forward methods for performing inference in statistical models based on sequential Monte Carlo and particle Markov chain Monte Carlo. We have shown that it is possible to efficiently and scalably implement this particular kind of inference strategy using existing, standard compilers and posix compliant operating system primitives.

What most distinguishes Probabilistic C from prior art is that it is highly compatible with modern computer architectures, all the way from operating systems to central processing units (in particular their virtual memory operations), and, further, that it delineates a future research program for scaling the performance of probabilistic programming systems to large scale problems by investigating systems optimizations of existing computer architectures. Note that this is distinct but compatible with approaches to optimizing probabilistic programming systems by compilation optimizations, stochastic hardware, and dependency tracking with efficient updating of local execution trace subgraphs. It may, in the future, be possible to delineate model complexity and hardware architecture regimes in which each approach is optimal; we assert that, for now, it is unclear what those regimes are or will be. This paper is but one step towards such a delineation.

Several interesting research questions remain: (1) Is it more sensible to write custom memory management and use threads than fork and processes as we have done? The main contribution of this paper is to establish a probabilistic programming system implementation against a standardized, portable abstraction layer. It might be possible to eke out greater performance by capitalizing on the fact modern architectures are optimised for parallel threads more so than parallel processes; however, exploiting this would entail implementing memory management de facto equivalent in action to fork which may lead to lower portability. (2) Would shifting architectures to small page sizes help? There is a bias towards large page size support in modern computer architectures. It may be that the system use characteristics of probabilistic programming systems might provide a counterargument to that bias or inspire the creation of tuned end-to-end systems. Forking itself is lightweight until variable assignment which usually require manipulations of entire page tables. Large pages require large amounts of amortisation in order to absorb the cost of copying upon stochastic variable assignment. Smaller pages could potentially yield higher efficiencies. (3) What characteristics of process synchronisation can be improved specifically for probabilistic programming systems? This is both a systems and machine learning question. From a machine learning perspective we believe it may be possible to construct efficient sequential Monte Carlo algorithms that do not synchronize individual threads at observe barriers and instead synchronize in a queue. On a systems level it begs questions about what page replacement strategies to consider; perhaps entirely changing the page replacement schedule to reflect rapid process rather than thread multiplexing across cores.

It is the case that probabilistic C on its own allows programmers to easily write models that are very difficult to reason about statistically. For this reason we are not, in this paper, making a claim about a new probabilistic programming language – rather, in a programming languages sense, we at most claiming an intermediate representation compilation target that implements a particular style of inference that is natively parallel and possible to optimise by system architecture choice. It is possible (as helpfully pointed out by Vikash Mansinghka) that compiler optimisation techniques such as checking to see whether or not the program is natively in "static single assignment" form (Appel, 1998) can help avoid statistically ambiguous programs being allowed (at compilation). This we leave as future work. Further (as helpfully pointed out by Noah Goodman), programming languages constructs such as delimited continuations (Felleisen, 1988) can be thought of as user level abstractions of fork, and as such might provide, in the context of statistically safer languages, similar functionality at the user rather than system level.

References

Andrieu, Christophe, Doucet, Arnaud, and Holenstein, Roman. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, 2010.

Appel, Andrew W. SSA is functional programming. *SIGPLAN notices*, 33(4):17–20, 1998.

Doucet, Arnaud, De Freitas, Nando, Gordon, Neil, et al. *Sequential Monte Carlo methods in practice*. Springer New York, 2001.

Felleisen, Mattias. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 180–190. ACM, 1988.

Goodman, Noah D., Mansinghka, Vikash K., Roy, Daniel M., Bonawitz, Keith, and Tenenbaum, Joshua B. Church: A language for generative models. In *In UAI*, pp. 220–229, 2008.

Kiselyov, Oleg and Shan, Chung-chien. Monolingual probabilistic programming using generalized coroutines. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence (UAI2009)*, 2009.

Mansinghka, Vikash, Selsam, Daniel, and Perov, Yura. Venture: an interactive, Turing-complete probabilistic programming platform, 2013. URL http://probcomp.csail.mit.edu/venture/.

Milch, Brian, Marthi, Bhaskara, Russell, Stuart, Sontag, David, Ong, Daniel L, and Kolobov, Andrey. BLOG: Probabilistic models with unknown objects. *Statistical relational learning*, pp. 373, 2007.

Open Group, The. IEEE Std 1003.1, 2004 Edition, 2004a. URL http://pubs.opengroup.org/onlinepubs/009695399.

Open Group, The. IEEE Std 1003.1, 2004 Edition, 2004b. URL http://pubs.opengroup.org/onlinepubs/009695399/functions/fork.html.

Pfeffer, Avi. IBAL: A probabilistic rational programming language. In *IJCAI*, pp. 733–740, 2001.

Pfeffer, Avi. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, pp. 137, 2009.

Smith, Jonathan M. and Maguire, Jr, Gerald Q. Effects of copy-on-write memory management on the response time of UNIX fork operations. *COMPUTING SYSTEMS*, 1(3):255–278, 1988.

Wingate, David, Stuhlmueller, Andreas, and Goodman, Noah D. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the 14th international conference on Artificial Intelligence and Statistics*, pp. 131, 2011.

Wood, Frank, van de Meent, Jan Willem, and Mansinghka, Vikash. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, 2014.