JAX Based Parallel Inference for Reactive Probabilistic Programming

Guillaume Baudart
DI ENS, École normale supérieure,
PSL University, CNRS, Inria
France

Louis Mandel
IBM Research
USA

Reyyan Tekin
DI ENS, École normale supérieure,
PSL University, CNRS, Inria
France

Abstract

ProbZelus is a synchronous probabilistic language for the design of reactive probabilistic models in interaction with an environment. Reactive inference methods continuously learn distributions over the unobserved parameters of the model from statistical observations. Unfortunately, this inference problem is in general intractable. Monte Carlo inference techniques thus rely on many independent executions to compute accurate approximations. These methods are expensive but can be parallelized.

We propose to use JAX to parallelize ProbZelus reactive inference engine. JAX is a recent library to compile Python code which can then be executed on massively parallel architectures such as GPUs or TPUs.

In this paper, we describe a new reactive inference engine implemented in JAX and the new associated JAX backend for ProbZelus. We show on existing benchmarks that our new parallel implementation outperforms the original sequential implementation for a high number of particles.

CCS Concepts: • **Computing methodologies → Parallel computing methodologies**; • **Mathematics of computing → Bayesian computation**; **Sequential Monte Carlo methods**; • **Software and its engineering → Data flow languages**.

Keywords: Probabilistic Programming, Reactive Programming, Streaming Inference, Parallel Computing, Compilation

ACM Reference Format:
Guillaume Baudart, Louis Mandel, and Reyyan Tekin. 2022. JAX Based Parallel Inference for Reactive Probabilistic Programming. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '22), June 14, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3519941.3535066

LCTES '22, June 14, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9266-2/22/06…$15.00
https://doi.org/10.1145/3519941.3535066

1 Introduction

Synchronous languages [5] were introduced to ease the design and implementation of real-time embedded systems. Designers use these languages to write high-level executable specifications that can be simulated, tested, and formally verified. Specialized compilers then generate *correct by construction* embedded code, i.e., code that preserves the semantics of the initial specification. For example, the synchronous language Scade [14] is now routinely used in industry to program airplane and train controllers.

Most embedded systems operate in an open environment that they only perceive through noisy sensors, e.g., accelerometers, camera. Interactions with autonomous agents, e.g., animals, humans, robots, add another level of uncertainty. However, synchronous languages offer limited support for modeling and reasoning about this uncertainty.

General purpose probabilistic programming languages have been introduced to describe models that explicitly manipulate uncertainty, and automatically infer distributions over parameters from statistical observations [7, 18, 22, 25, 26]. These languages are based on Bayesian inference where a *prior* belief on parameters distributions is updated using concrete data to obtain a *posterior* distribution. In this line, ProbZelus is a recent language [2] combining reactive constructs from synchronous dataflow languages (discrete global clock, parallel composition, hierarchical automata) with probabilistic constructs (sampling from distribution and conditioning a model on observed data). ProbZelus can thus be used to design reactive applications involving probabilistic models, e.g., a robot controller which continuously estimates its position to adapt its trajectory [1].

Given a probabilistic model the inference problem is in general intractable. Monte Carlo inference techniques thus rely on multiple independent executions, called *particles*, to approximate the posterior distribution [16]. These methods are expansive but can be parallelized.

ProbZelus is compiled to sequential code that relies on imperative memory updates which cannot be used for parallel execution. In this paper, we propose to use JAX to parallelize ProbZelus runtime. JAX is a recent library that can seamlessly compile code written in a purely functional subset of Python that can then run on massively parallel architectures such as GPUs or TPUs [11]. JAX comes with an impressive numerical library that can be leveraged to implement

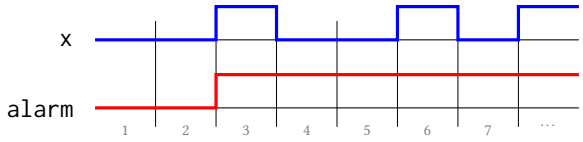


Figure 1. Sample execution trace for the node `watch`.

ProbZelus inference engine. In the future, additional JAX features such as automatic differentiation could be used to implement more advanced inference techniques.

In this paper, we use JAX to parallelize the execution of reactive probabilistic applications. We present the following contributions.

- A new JAX based parallel inference runtime (Section 3).
- A new compiler from ProbZelus to JAX (Section 4).
- We evaluate the performance of our proposal on a set of existing benchmarks (Section 5).

The source code is available in the ACM Digital Library [3] and at https://github.com/rpl-lab/lctes22-zlax.

2 Background

In this section we recall the basics of synchronous programming and probabilistic programming on an example.

2.1 Reactive Probabilistic Programming

Synchronous programming. ProbZelus extends the synchronous language Zelus[1] [10] with probabilistic constructs. Zelus is a dataflow synchronous language à la Lustre [20]. A program defines a set of stream processors, called *nodes*. All streams progress *synchronously*, i.e., time proceeds by discrete logical steps, and at each step, a node computes the value of each stream depending on its inputs and possibly previously computed values. This programming style is a natural fit to describe block-diagram, a classic notation for embedded reactive system [14].

For instance, the following node raises an alarm when the Boolean input stream is set to true.

```
node watch x = alarm where
  rec automaton
  | Wait → do alarm = false unless x then Ring
  | Ring → do alarm = true done
```

The node `watch` computes the output stream `alarm` from the input stream `x`. The behavior of this node is defined with a two-states automaton. In the initial state `Wait`, the output `alarm` is always `false`. When `x` is set to `true`, the transition `unless x` is activated and the automaton enters the state `Ring`. The output `alarm` is then always `true`. Figure 1 shows a sample execution trace for the node `watch`.

[1] https://zelus.di.ens.fr

Probabilistic programming. The goal of Bayesian inference is to estimate the distribution of a parameter θ given a series of observations $\mathbf{x}$, the *posterior distribution* $p(\theta \mid \mathbf{x})$, from an initial belief, the *prior distribution* $p(\theta)$.

$$p(\theta \mid \mathbf{x}) = \frac{p(\theta)\, p(\mathbf{x} \mid \theta)}{p(\mathbf{x})} \qquad \text{(Bayes, 1763)}$$

To describe probabilistic models, recent probabilistic languages typically extend general purpose languages with three specialized constructs: `sample`, `observe`, and `infer`. `theta = sample(d)` introduces a random variable `theta` with a prior distribution `d`; `observe(d, x)` conditions the model assuming the observation `x` was sampled from the distribution `d`; `infer m x` computes the posterior distribution of output values of a model given the input data `x`.

A classic introductory example is to compute the distribution of the bias of a coin from a series of independent tosses. At each step $n \in \mathbb{N}$ we observe the results of a coin toss: $x_n = \mathit{True}$ (head) or $x_n = \mathit{False}$ (tail). We assume that every toss follows a Bernoulli distribution of parameter θ: $p(x_n \mid \theta) = \theta$ and $p(\overline{x_n} \mid \theta) = 1 - \theta$. The goal is to estimate the bias θ from the observations $(x_n)_{n\in\mathbb{N}}$, i.e., $p(\theta \mid x_0, x_1, x_2, \dots)$.

The following program implement this model in ProbZelus.

```
proba coin x = theta where
  rec init theta = sample (uniform_float (0., 1.))
  and () = observe (bernoulli theta, x)
```

Compared to a classic Zelus node, a probabilistic model is introduced with the keyword `proba`. Initially we have no information, all possible bias are equiprobable. The prior distribution is thus a uniform distribution over $[0, 1]$ ($\theta = 0.5$ corresponds to a fair coin, $\theta = 0$ is a coin that always falls on tail). We assume that the parameter θ is constant over time (keyword `init`). At each step, we use the `observe` construct to condition the model on the fact that the observation `x` follows a Bernoulli distribution with parameter θ.

2.2 Inference in the Loop

The `infer` construct is a higher-order node that takes as argument a model node and an input stream. At each step, the output of `infer` is the distribution of output values given the inputs observed so far. Inference is a synchronous process that never stops and can be executed *in the loop* with classic deterministic nodes. For instance, the following program combines the previous `coin` model and the `watch` node to implement a cheater detector.

```
node cheater_detector x = a where
  rec theta_dist = infer coin x
  and m, s = stats_float theta_dist
  and a = watch ((m < 0.2 || 0.8 < m) && (s < 0.01))
```

The stream `theta_dist` is the estimated distribution for the parameter θ given the first observations $x_0, x_1, \dots, x_n$. At

each step, we compute the mean `m` and standard deviation `s` of this distribution. The node `watch` defined above raise the alarm `a` as soon as we are reasonably confident that the bias is over 0.8 or below 0.2.

2.3 ProbZelus

The syntax, typing rules, and semantics of ProbZelus are formally defined in [2]. In this section we briefly recall the main ideas that are required to understand our contributions.

Syntax. The syntax of a kernel of ProbZelus is the following. Missing constructs, e.g., hierarchical automata, can be compiled to this kernel via a series of source-to-source transformations.

$$
\begin{array}{rcl}
d & ::= & \texttt{node}\ f\ x = e \mid \texttt{proba}\ f\ x = e \mid d\ d \\
e & ::= & c \mid x \mid (e, e) \mid op(e) \mid f(e) \mid \texttt{last}\ x \mid e\ \texttt{where rec}\ E \\
 & & \mid \texttt{present}\ e\ \texttt{->}\ e\ \texttt{else}\ e \mid \texttt{reset}\ e\ \texttt{every}\ e \\
 & & \mid \texttt{sample}(e) \mid \texttt{observe}(e, e) \mid \texttt{infer}(f(e)) \\
E & ::= & x = e \mid \texttt{init}\ x = c \mid E\ \texttt{and}\ E
\end{array}
$$

A program is a sequence of deterministic (`node`) or probabilistic (`proba`) nodes declarations. The body of a node is an expression. Expressions comprise constants, variables, pairs, external operator applications (e.g., `+`, `-`, `mean`), function or node applications, logical delays (`last` x returns the value of x at the previous step), or local definitions with a set of mutually recursive equations. An equation $x = e$ defines the value of the variable x and an equation `init` $x = c$ defines the initial (constant) value of x. In addition, the kernel comprises two control structures: `present` x `->` e_1 `else` e_2 lazily computes e_1 or e_2 depending on the current value of x; `reset` e_1 `every` e_2 resets the values of the `last` x expressions in e_1 to their initial values defined by `init` $x = c$ equations. Finally, this language is extended with the probabilistic expressions `sample`(e), `observe`(e_1, e_2), and `infer`$(f(e))$.

Scheduling. At compile time, equations are simplified and scheduled according to data dependencies. Initializations `init` x_j `=` c_j are grouped at the beginning and an equation $x_j = e_j$ must appear before $x_i = e_i$ if e_j depends on x_i outside a `last` operator. Programs that cannot be statically scheduled are rejected by a dedicated type system [4]. To simplify the presentation, we assume that programs are scheduled. We also assume that all the variables introduced with `init` are also defined by an equation. The compiler can always add an equation x_i `=` `last` x_i to satisfy this assumption.

Semantics. The semantics of ProbZelus is defined in a *co-iterative* framework [12]. Given an environment γ mapping variable names to their values, deterministic expressions define states machines characterized by an initial state $[\![e]\!]^{\text{init}}_{\gamma}$ of type S, and a transition function $[\![e]\!]^{\text{step}}_{\gamma}$ of type $S \to T \times S$. From the current state, firing the transition function returns an output and the next state. The corresponding stream is obtained by repeatedly firing the transition function from the initial state. For instance, the semantics of a variable and the lazy `present` construct are defined as follows:

$$
\begin{array}{l}
[\![x]\!]^{i}_{\gamma} = () \\
[\![x]\!]^{s}_{\gamma} = \lambda s.\ (\gamma(x), s) \\
[\![\texttt{present}\ e\ \texttt{->}\ e_1\ \texttt{else}\ e_2]\!]^{\text{init}}_{\gamma} = ([\![e]\!]^{\text{init}}_{\gamma}, [\![e_1]\!]^{\text{init}}_{\gamma}, [\![e_2]\!]^{\text{init}}_{\gamma}) \\
[\![\texttt{present}\ e\ \texttt{->}\ e_1\ \texttt{else}\ e_2]\!]^{\text{step}}_{\gamma} = \\
\quad \lambda(s, s_1, s_2).\ \mathit{let}\ v, s' = [\![e]\!]^{\text{step}}_{\gamma}(s)\ \mathit{in} \\
\quad\quad \mathit{if}\ v\ \mathit{then}\ \mathit{let}\ v_1, s_1' = [\![e_1]\!]^{\text{step}}_{\gamma}(s_1)\ \mathit{in}\ (v_1, (s', s_1', s_2)) \\
\quad\quad\quad \mathit{else}\ \ \mathit{let}\ v_2, s_2' = [\![e_2]\!]^{\text{step}}_{\gamma}(s_2)\ \mathit{in}\ (v_2, (s', s_1, s_2'))
\end{array}
$$

The transition function of a variable always returns the corresponding value in the environment, and the state is empty. The transition function of `present` e `->` e_1 `else` e_2 lazily executes one of the two branches (e_1 or e_2) depending on the value of e. The state of `present` is composed of the state of the three sub-expressions.

Probabilistic expressions are also characterized by an initial state $\{\![e]\!\}^{\text{init}}_{\gamma}$ and a transition function $\{\![e]\!\}^{\text{step}}_{\gamma}$, but the transition function returns a *measure* which associates a positive score to a set of possible outcomes, i.e., a set of pairs (output, next state). The type of the transition function is thus $S \to \Sigma_{T \times S} \to [0, \infty)$ where $\Sigma_{T \times S}$ is the σ-algebra over $T \times S$, i.e., the set of measurable sets over $T \times S$.

The `infer` operator returns a deterministic expression (a distribution), from a probabilistic model f and a stream of observations e. At each step, `infer` computes the distribution of output values and a distribution of possible next states.

$$
\begin{array}{l}
[\![\texttt{infer}(f(e))]\!]^{\text{init}}_{\gamma} = \lambda U.\ \delta_{[\![f(e)]\!]^{i}_{\gamma}}(U) \\
[\![\texttt{infer}(f(e))]\!]^{\text{step}}_{\gamma} = \\
\quad \lambda\sigma.\ \mathit{let}\ \mu = \lambda U.\ \int_S \sigma(ds) \{\![f(e)]\!\}^{\text{step}}_{\gamma}(s)(U)\ \mathit{in} \\
\quad\quad \mathit{let}\ \nu = \lambda U.\ \mu(U)/\mu(\top)\ \mathit{in} \\
\quad\quad (\pi_{1*}(\nu), \pi_{2*}(\nu))
\end{array}
$$

The initial state $[\![\texttt{infer}(f(e))]\!]^{\text{init}}$ is the Dirac delta measure on the initial state of $f(e)$. The transition function integrates the measure defined by the model $\{\![f(e)]\!\}^{\text{step}}_{\gamma}$ over all possible states. The resulting measure μ is then normalized into a distribution $\nu : T \times S$ `dist` ($\top$ denotes the entire space) that is then split into a pair of marginal distributions over results and next states using the pushforward measures by the projections π_1 and π_2.

3 Inference Engine

Computing the posterior distribution of a probabilistic model is in general intractable. Most probabilistic programming languages thus rely on approximate inference algorithms. Monte Carlo methods estimates a distribution from multiple independent random simulations [16].

In this section we formalize two approximate inference methods for ProbZelus models: *importance sampling* and *particle filtering*. Compared to [2] where inference is only

$$\{\![c]\!\}_\gamma^{\text{step}} = \lambda s, w.\ (c, s, w)$$

$$\{\![x]\!\}_\gamma^{\text{step}} = \lambda s, w.\ (\gamma(x), s, w)$$

$$\{\![\texttt{sample(}e\texttt{)}]\!\}_\gamma^{\text{step}} =$$
$$\lambda s, w.\ \textit{let}\ d, s', w' = \{\![e]\!\}_\gamma^{\text{step}}(s, w)\ \textit{in}\ (\text{draw}(d), s', w')$$

$$\{\![\texttt{observe(}e_1\texttt{,}e_2\texttt{)}]\!\}_\gamma^{\text{step}} =$$
$$\lambda(s_1, s_2), w.$$
$$\quad \textit{let}\ \mu, s_1', w_1 = \{\![e_1]\!\}_\gamma^{\text{step}}(s_1, w)\ \textit{in}$$
$$\quad \textit{let}\ v, s_2', w_2 = \{\![e_2]\!\}_\gamma^{\text{step}}(s_2, w_1)\ \textit{in}\ ((), (s_1', s_2'), w_2 * \mu_{\text{pdf}}(v))$$

$$\{\![\texttt{present}\ e\ \texttt{->}\ e_1\ \texttt{else}\ e_2]\!\}_\gamma^{\text{step}} =$$
$$\lambda(s, s_1, s_2), w.$$
$$\quad \textit{let}\ v, s', w' = \{\![e]\!\}_\gamma^{\text{step}}(s, w)\ \textit{in}$$
$$\quad \textit{if}\ v\ \textit{then let}\ v_1, s_1', w_1 = \{\![e_1]\!\}_\gamma^{\text{step}}(s_1, w')\ \textit{in}\ (v_1, (s', s_1', s_2), w_1)$$
$$\qquad \textit{else}\ \ \textit{let}\ v_2, s_2', w_2 = \{\![e_2]\!\}_\gamma^{\text{step}}(s_2, w')\ \textit{in}\ (v_2, (s', s_1, s_2'), w_2)$$

$$\{\![\texttt{reset}\ e_1\ \texttt{every}\ e_2]\!\}_\gamma^{\text{step}} =$$
$$\lambda(s_0, s_1, s_2), w.$$
$$\quad \textit{let}\ v_2, s_2', w_2 = \{\![e_2]\!\}_\gamma^{\text{step}}(s_2, w)\ \textit{in}$$
$$\quad \textit{let}\ v_1, s_1', w_1 = \{\![e_1]\!\}_\gamma^{\text{step}}(\textit{if}\ v_2\ \textit{then}\ (s_0, w_2)\ \textit{else}\ (s_1, w_2))\ \textit{in}$$
$$\quad (v_1, (s_0, s_1', s_2'), w_1)$$

$$\left\{\!\!\left[\begin{array}{l} e\ \texttt{where rec init}\ x_1\ \texttt{=}\ c_1\ \ \texttt{...}\ \ \texttt{and init}\ x_k\ \texttt{=}\ c_k \\ \qquad\quad \texttt{and}\ y_1\ \texttt{=}\ e_1\ \ \texttt{...}\ \ \texttt{and}\ y_n\ \texttt{=}\ e_n \end{array}\right]\!\!\right\}_\gamma^{\text{step}} =$$
$$\lambda((m_1, \ldots, m_k), (s_1, \ldots, s_n), s), w.$$
$$\quad \textit{let}\ \gamma_1 = \gamma[m_1/x_1\texttt{\_last}]\ \textit{in}\ \ldots\ \textit{let}\ \gamma_k = \gamma_{k-1}[m_k/x_k\texttt{\_last}]\ \textit{in}$$
$$\quad \textit{let}\ v_1, s_1', w_1 = \{\![e_1]\!\}_{\gamma_k}^{\text{step}}(s_1, w)\ \textit{in let}\ \gamma_1' = \gamma_k[v_1/y_1]\ \textit{in}\ \ldots$$
$$\quad \textit{let}\ v_n, s_n', w_n = \{\![e_n]\!\}_{\gamma_{n-1}'}^{\text{step}}(s_n, w_{n-1})\ \textit{in let}\ \gamma_n' = \gamma_{n-1}'[v_n/y_n]\ \textit{in}$$
$$\quad \textit{let}\ v, s', w' = \{\![e]\!\}_{\gamma_n'}^{\text{step}}(s)\ \textit{in}$$
$$\quad (v, ((\gamma_n'[x_1], \ldots, \gamma_n'[x_k]), (s_1', \ldots, s_n'), s'), w')$$

Figure 2. Operational semantics of probabilistic expressions.

defined on the compiled imperative code, for each inference method we define the operational semantics directly on the dataflow source code. We then show how the runtime can be parallelized using JAX.

3.1 Sampler

The operational semantics of a probabilistic model for Monte Carlo methods is a *sampler*. Initial states are the same as in the ideal semantics presented in Section 2.3, but transition functions return a random sample together with a score measuring the quality of this sample.

The transition function of a probabilistic expression is of type $S \times [0, \infty) \to T \times S \times [0, \infty)$. Given the current state and the current score, the transition function returns an output, the next state and an updated score. The definition of the transition functions are presented in Figure 2. Constants and variables accesses return the expected value without changing the score. `sample(d)` draws a random sample from the distribution `d` without changing the score. `observe(d, x)` multiplies the score by the likelihood of the observation `x` w.r.t. the distribution `d` (i.e., the value of the density function $\texttt{d}_{\text{pdf}}$ at `x`) and returns the empty value `()`. For other expressions, the transition function simply computes the next state and propagates the score following dependencies order between sub-expressions.

For the `where rec` expression, the variables x_is are a subset of the variables y_js. The state of this expression contains the state of all the sub-expressions, and the value of all local variables at the previous step $(m_1, \ldots, m_k)$. The transition function first updates the environment γ with a set of fresh variables x_i`_last` initialized with the values m_i. This environment is then extended with the definitions of the variables y_i by executing all the sub-expressions while propagating the score. Then the main expression e is executed in the final environment. The next state contains the new value of all the initialized variable that will be used to start the next step.

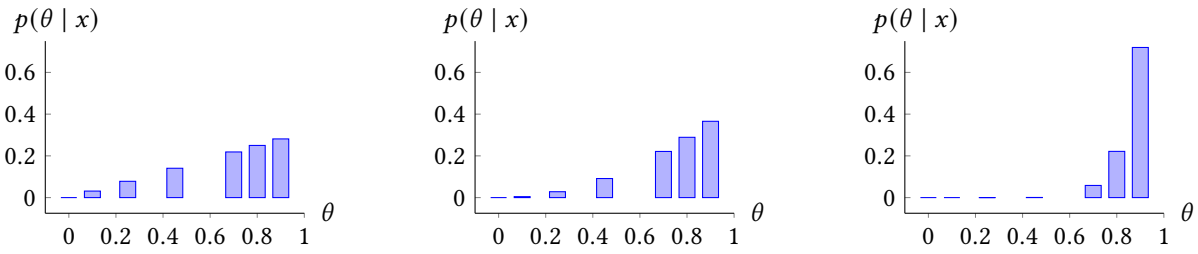
3.2 Importance Sampling

The most simple sampler based inference method launches N independent executions of the model, called *particles*. At each step, each particle executes one step of the sampler to compute a triple (results, next state, score). Scores are then normalized to obtain a categorical distribution, i.e., a weighted discrete distribution over pairs (result, next state).

$$[\![\texttt{infer(}f\texttt{(}e\texttt{))}]\!]_\gamma^{\text{init}} = [([\![f(e)]\!]_\gamma^{\text{init}}, 1)]_{1 \le i \le N}$$
$$[\![\texttt{infer(}f\texttt{(}e\texttt{))}]\!]_\gamma^{\text{step}} =$$
$$\lambda s.$$
$$\quad \textit{let}\left[(o_i, s_i', w_i') = \textit{let}\ s_i, w_i = s[i]\ \textit{in}\ \{\![f(e)]\!\}_\gamma^{\text{step}}(s_i, w_i)\right]_{1 \le i \le N} \textit{in}$$
$$\quad \textit{let}\ \mu = \lambda U.\ \textstyle\sum_{1 \le i \le N} \overline{w_i'} * \delta_{o_i}(U)\ \textit{in}$$
$$\quad \mu, [(s_i', w_i')]_{1 \le i \le N}$$

The state of the `infer` operator is an array initialized with N copies of the sampler initial state with the initial score 1. At each step, each particle retrieves its current state and current score in the array to execute one step of the sampler. Scores are then normalized to obtain the resulting distribution μ and the array is updated for the next step ($\overline{w_i} = w_i / \sum_{j=1}^N w_j$ denotes the normalized scores). Compared to the ideal semantics of `infer` presented in Section 2.3, importance sampling approximates the intractable integral with a discrete sum over the array of particles.

Consider the `coin` model of Section 2.1. At the initial step, the first equation `init theta = sample (uniform_float (0.,1.))` draws a set of possible values for the parameter `theta`. Then, at each step, the first equation does not change (operator `init`) but the second one `() = observe (bernoulli theta, x)` updates the score for each possible `theta` value as follows: $w' = w * \textit{Bernoulli}(\theta)_{\text{pdf}}(x) = w * \theta^x (1-\theta)^{(1-x)}$. We can then normalize the scores to obtain the posterior distributions of results at each step. Figure 3 shows one possible execution when observations are always head/`true`. As expected, over time values closer to 1 (a coin that always returns head) are the most probable.



(a) step 1 **(b)** step 2 **(c)** step 10

Figure 3. Importance sampling with 7 particles for the `coin` model. We observe head/`true` at each step.

Figure 4 illustrates the precision of importance sampling for an increasing number of particles on the `coin` example. On this simple example 1000 particles are enough to obtain a reasonable approximation of the theoretical distribution: $Beta(1, 10 + 1)$. For more complex examples with multi-dimensional parameters, the number of required particles quickly increases. However, since particles are independent from each other, we can parallelize computations.

3.3 JAX Based Parallelization

JAX[2] is a recent library that can seamlessly compile code written in a purely functional subset of Python that can then run on massively parallel architectures such as GPUs or TPUs [11]. During execution, a *Just-In-Time* (JIT) compiler specializes Python functions that are then compiled by XLA,[3] the Google high-performance compiler for GPUs and TPUs.

`vmap` *operator.* The JAX operator `vmap` automatically vectorizes a function for data parallel computations. For tree-like data structures, this operator preserves the input/output structures and only vectorizes the leafs. For example, given a function such that `f(0) = ((1, 2), 3)` we have:

`vmap(f)([0, 0, 0]) = (([1,1,1], [2,2,2]), [3,3,3])`.

More generally, if inputs are of type `float t_in` and outputs are of type `float t_out` where `float t_in` and `float t_out` are static tree-like data structures where all leafs are of type `float`, the type of `vmap` is:

```
val vmap:
  (float t_in → float t_out) → float array t_in
    → float array t_out
```

In ProbZelus, this property is paramount to vectorize the state of the particles which contains the state of all sub-expressions in the form of nested tuples (cf. Section 2.3). Using the `vmap` operator, we can implement a higher-order ProbZelus node `zmap` which launches N parallel instances of a node as follows:

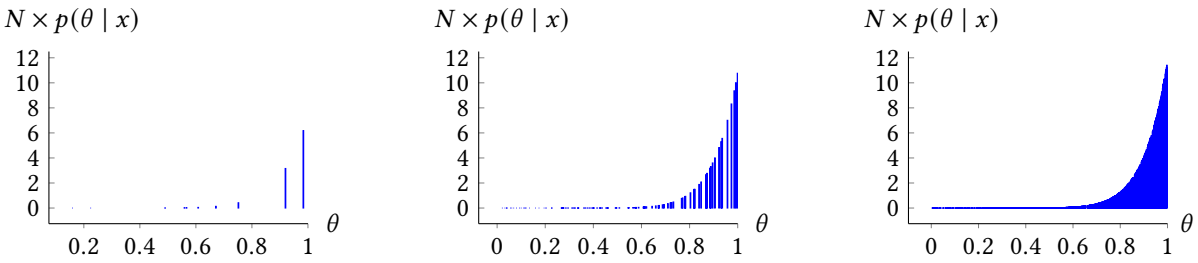
[2] https://github.com/google/jax
[3] https://www.tensorflow.org/xla

```
from jax import vmap
from jax.numpy import empty
class zmap(Node):
  def __init__(self, f, n):
    f = f()
    self.f_vinit = vmap(lambda _: init(f))
    self.f_vstep = vmap(step(f))
    self.n = n
  def init(self)
    s_init = self.f_vinit(empty(self.n))
    return s_init
  def step(self, s, i):
    o, s = self.f_vstep(s, i)
    return s, o
```

ProbZelus nodes must implement the `Node` class which imposes the definition of the two methods `init` (memory allocation) and `step` (transition function). The instantiation of this class creates `f_vinit` and `f_vstep` the vectorized versions of `init` and `step` for the node `f` given as an argument. The `init` method of `zmap` applies `f_vinit` on an empty array of size N to allocate the initial state with the correct dimension. The transition function of `zmap` applies `f_vstep` on the (vectorized) current state `s` and an input array `i` of the same dimension.

The implementation of the importance sampler follows the same scheme. The initial state contains N copies of the initial state of `f` associated with the initial score 1. The transition function unpacks the current state into a pair (state, score) that can be used to vectorize the execution of the sampler. Results are then normalized to obtain a distribution.

```
class infer_importance(Node):
  def __init__(self, f, n):
    f = f()
    self.f_vinit = vmap(lambda _: init(f), 1.0)
    self.f_vstep = vmap(step(f))
    self.n = n
  def init(self):
    return self.f_vinit(empty(self.n))
```



(a) 10 particles **(b)** 100 particles **(c)** 1000 particles

Figure 4. Importance sampling accuracy as a function of the number of particles N after 10 steps for the `coin` example.

```
def step(self, s, obs):
  s_in, w_in = s
  o, s_out, w_out = self.f_vstep(s_in, w_in, obs)
  mu = normalize(o, w_out)
  return mu, (s_out, w_out)
```

Remark. JAX random number generator (used to implement the `sample` construct) requires an explicit random seed. To simplify the presentation, we ignored this detail. In practice, transition functions take an additional argument *key* that can be grouped with the score to form a single argument *proba* containing all the required information to execute the probabilistic transition functions (cf. Section 4.1).

3.4 Particle Filter

Importance sampling is relatively accurate to infer constant parameters from a series of observations, as in the `coin` example. Unfortunately, this method quickly shows its limitations on models where parameters vary over time. For instance, the following model implement a Hidden Markov Model (HMM) to estimate the current position of a moving object from noisy observations.

```
proba hmm obs = x where
  rec x = sample (gaussian (0. → pre x), speed)
  and () = observe (gaussian (x, noise), obs)
```

At each step, we assume that the current position is not too far from the previous position, i.e., the current position follows a normal distribution centered on `0. → pre x` (the previous value of `x` initialized with 0). `speed` and `noise` are global constants.

For such models, importance sampling is equivalent to a random walk: at each step, we draw the current value of `x` at random. The probability that a random trajectory matches a series of observation quickly collapse to 0. Estimations are thus unusable after 2 or 3 steps (Figure 5a).

Resampling. To mitigate this issue, a particle filter introduces a resampling step. The number of particles remains constant over time, but at each time step, the least likely particles are discarded and the most likely particles are duplicated. In other words, the particles array is re-centered on the current observation.

$$[\![\texttt{infer}(f(e))]\!]_\gamma^{\text{init}} = [([\![f(e)]\!]_\gamma^{\text{init}}, 1)]_{1\le i\le N}$$

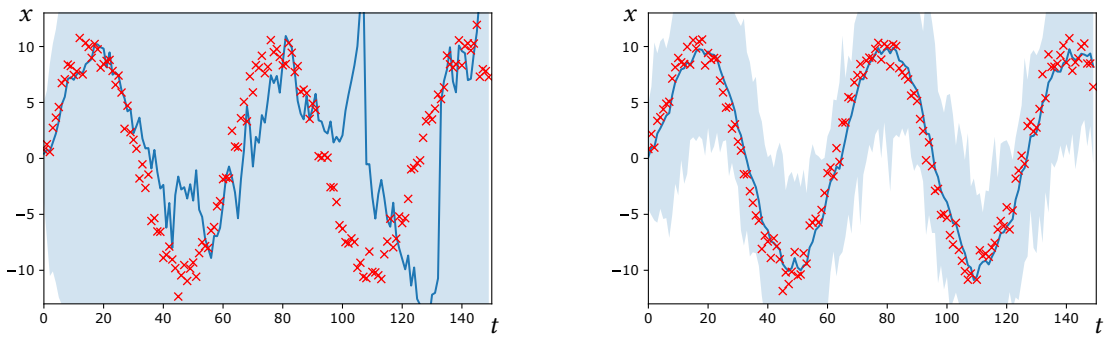
$$[\![\texttt{infer}(f(e))]\!]_\gamma^{\text{step}} =$$

$$\lambda s.\ let\ \left[(o_i, s_i', w_i') = let\ s_i, w_i = s[i]\ in\ \{[\![f(e)]\!]_\gamma^{\text{step}}(s_i, w_i)\right]_{1\le i\le N} in\ let\ \mu = \lambda U.\ \textstyle\sum_{1\le i\le N} \overline{w_i'} * \delta_{(o_i,s_i)}(U)\ in\ \pi_{1*}(\mu), [(\text{draw}(\pi_{2*}(\mu)), 1)]_{1\le i\le N}$$

The semantics of the `infer` operator is similar to the importance sampler. At each step, each particle executes one step of the sampler. Scores are then normalized to obtain a distribution over pairs (results, next state). As in the ideal semantics presented in Section 2.3, this distribution is then split into a pair of marginal distributions. We can then sample N values in the distribution of next state to compute the new particles array and reset the score of each particle to 1.

The implementation of the particle filter is similar to the `infer_importance` node. We use the `vmap` operator to vectorize the execution of the sampler. The `normalize` function returns a categorical distribution that can be sampled with the `sample` method.

```
class infer_pf(Node):
  def __init__(self, f, n):
    f = f()
    self.f_vinit = vmap(lambda _: (init(f), 1.0)
    self.f_vstep = vmap(step(f))
    self.n = n
  def init(self):
    return self.f_vinit(np.empty(self.n))
  def step(s, obs):
    s_in, w_in = s
    o, s_out, w_out = self.f_vstep(s_in, w_in, obs)
    mu = normalize(o, w_out)
    s_dist = normalize(s_out, w_out)
    s_out = s_dist.sample(sample_shape=self.n)
    return mu, (s_out, np.ones(self.n))
```



(a) importance sampling

(b) particle filter

Figure 5. Estimations for the `hmm` example with 100 particles on a noisy sinus after 150 steps. Noisy observations are shown in red, the blue line is the estimated mean, and the blue zone contains all the particles (min/max interval at each step).

Figure 5b shows the result of the particle filter on the `hmm` model. We observe that this inference method returns accurate estimations for a reactive probabilistic model where parameters change over time.

Remark. In practice, the developer has to pick the most suitable inference method for a given model.

4 Compiling ProbZelus to JAX

We now need to compile ProbZelus models to Python/JAX code that can be called by the nodes `infer_importance` and `infer_pf`. In this section we thus present a new JAX backend for ProbZelus.

Synchronous language compilers translate dataflow programs to imperative code. The memory required to run the program is statically allocated before execution and updated with side-effects. This compilation scheme, well suited to low-level languages used for embedded systems, guarantees a bounded memory execution. Unfortunately, JAX only accepts purely functional code which is easier to parallelize.

Our work is based on the Zelus [9] compiler which has the following architecture. (1) The source program is first analyzed and rewritten by a series of source-to-source transformations to a subset of the source language. These analyses and transformations include typing, causality analysis, hierarchical automata reduction, equations normalization, common sub-expressions elimination, or dead code removal. (2) Mutually recursive equations are then scheduled to satisfy data dependencies (cf. Section 2.3) and the program is translated into OBC, an intermediate imperative language. (3) The OBC code is compiled to imperative OCaml code.

To benefit as much as possible from this toolchain, we only change the last step. We translate the OBC code to μF, a purely functional language, which is then compiled to JAX. Compared to [2] where μF is only used to express the semantics but does not correspond to the generated code, here μF is the compilation target language.

4.1 Intermediate Languages

The OBC language. The intermediate language OBC (Object Based Code) was introduced to compile a synchronous language à la Lustre to imperative code (e.g., C) [6]. OBC is for instance used in the Heptagon [19], Vélus [8], and Zelus compilers (with minor differences for each of these compilers). In OBC, a stream function is characterized by a state and a transition function which perform in place imperative updates to the state. The syntax of OBC extended with probabilistic constructs is the following:

$$
\begin{array}{rcl}
\mathit{program} & ::= & d^* \\
d & ::= & \texttt{machine proba}^?\, m = \\
 & & \quad \texttt{memory}\ (x, \ldots, x) \\
 & & \quad \texttt{instances}\ (o : m, \ldots, o : m, \\
 & & \qquad\qquad o : \texttt{infer}(m), \ldots, o : \texttt{infer}(m)) \\
 & & \quad \texttt{reset}() = S \\
 & & \quad \texttt{step}(p)\ \texttt{returns}(p) = S \\
S & ::= & \texttt{var}\ x\ \texttt{in}\ S \mid x := e \mid \texttt{state}(x) := e \mid S\ ;\ S \mid \texttt{skip} \\
 & & \mid \texttt{match}\ x\ \texttt{with} \mid C \texttt{ -> } S \ldots \mid C \texttt{ -> } S \\
 & & \mid o.\texttt{reset} \mid p := o.\texttt{step}(e) \\
 & & \mid p := o.\texttt{pstep}(e) \mid p := \texttt{sample}(e) \mid \texttt{observe}(e, e) \\
e & ::= & c \mid x \mid \texttt{state}(x) \mid \mathit{op}(e) \\
p & ::= & x \mid (p, p)
\end{array}
$$

A program is a series of machine declarations (or classes). The optional `proba` annotation indicates a probabilistic model. A machine m comprises four fields: (1) `memory`: the state of the transition function, (2) `instances`: the machines used in m (node applications), (3) `reset`: the method to reset the state, and (4) `step`: the transition function that takes an

input, updates the state, and returns an output. Instances are annotated by the type m of the machine or `infer(`m`)` for instances of the `infer` operator.

A statement can be a declaration of a mutable local variable (`var` x `in` S), a local variable update (x `:=` e), a state variable update (`state(`x`)` `:=` e), a sequence of statements (S ; S), a statement with no effect (`skip`), a control structure (`match`/`with`), or a method call (`step` or `reset`). OBC is also extended with the probabilistic constructs `sample`, `observe`, and a method `pstep` which corresponds to a call to the `step` method of a probabilistic machine. An expression can be a constant (c), a local variable access (x), a state variable access (`state(`x`)`), or an operator application.

The μF language. μF is a first-order purely functional language [2]. The syntax of μF is defined by the following grammar:

$$
\begin{array}{rcl}
\mathit{program} & ::= & d^* \\
d & ::= & \texttt{val}\ p = e \\
 & | & \texttt{val}\ m = \texttt{stream}\ \{\ \texttt{init} = e\ ;\ \texttt{step}(p,p) = e\ \} \\
e & ::= & c \mid x \mid (e,e) \mid op(e) \mid f(e) \\
 & | & \texttt{match}\ x\ \texttt{with} \mid C \texttt{->}\ e \ldots \mid C \texttt{->}\ e \\
 & | & \texttt{let}\ p = e\ \texttt{in}\ e \mid \texttt{init}(m) \mid \texttt{unfold}(x,e) \\
 & | & \texttt{sample}(\mathit{proba},e) \mid \texttt{observe}(\mathit{proba},e,e) \\
 & | & \texttt{infer}(m)
\end{array}
$$

A program is a sequence of values and stream functions declarations. A stream function m comprises an initial state (`init`) and a transition function (`step`). A transition function takes a state and an input as arguments and returns an output and a new state. An expression can be a constant, a variable, a pair, an operator application, a function call, a conditional, or a local definition. The expression `init(`m`)` returns a new instance m whose state is the initial state. The expression `unfold(`x`,`e`)` executes one step of the instance x using the input e, and returns the next element in the stream and a new instance with an updated state.

The transition function of a probabilistic machine adds to its inputs/outputs a value *proba* which represents the state of the probabilistic model, e.g., the particles score, or the keys for the JAX random number generator (see Section 3.3). This *proba* value is passed as an argument, updated, and returned by the probabilistic operators `sample` and `observe`. Finally, the `infer` operator instantiates the inference operator on a probabilistic model.

4.2 Compilation

Compiling OBC to μF. The compilation function explicits its state manipulations to turn imperative OBC code into functional μF code. Consider the following machine m:

```
machine m =
    memory (x_1, ..., x_n)
    instances (o_1 : m_1, ..., o_t : m_t,
               o_{t+1} : infer(m_{t+1}), ..., o_k : infer(m_k))
    reset() = S
    step(p) returns(p) = S
```

The compilation of such a machine m produces a stream function declaration (`stream`). The compilation of the `reset` produces the `init` field and the compilation of the `step` method produces the `step` field.

$$
\begin{array}{l}
C(\texttt{reset}() = S) \ = \ \texttt{init} = C_{x_1,\ldots,x_n,o_1,\ldots,o_k}(S) \\
C(\texttt{step}(p_{input})\ \texttt{returns}(p_{output})) \ = \\
\quad \texttt{step}(x_1,\ldots,x_n,o_1,\ldots,o_k,p_{input}) = C_{(p_{output},x_1,\ldots,x_n,o_1,\ldots,o_k)}(S)
\end{array}
$$

For a probabilistic machine, the *proba* variable is added to the inputs and outputs of the `step` function.

$$
\begin{array}{c}
C(\texttt{step}(p_{input})\ \texttt{returns}(p_{output})) = \\
\texttt{step}(x_1,\ldots,x_n,o_1,\ldots,o_k,(\mathit{proba},p_{input})) = \\
C_{((p_{output},\mathit{proba}),x_1,\ldots,x_n,o_1,\ldots,o_k)}(S)
\end{array}
$$

The compilation function for OBC statements $C_p(S)$ is defined in Figure 6. This function is parameterized by the set p of variables that may be updated by the evaluation of the statement S and returns the value of these variables. Compiling `skip` ($C_p($`skip`$)$) simply returns p. The compilation of an assignment ($C_p(x := e)$) respects the invariant that the variable x is in p. The expression `let` $x = C(e)$ `in` p thus hides the previous value of x and returns its new value. Compiling the o`.reset` method instantiates a new state. Depending on the instance declaration o, the state is allocated with the `init` or the `infer` operator. The compilation of probabilistic machines, `sample`, and `observe`, explicitly updates the probabilistic state. Otherwise, the compilation function for expressions C is mostly the identity function. Note that in μF, the state is treated like a local variable $C(\texttt{state}(x)) = x$.

Remark. This compilation function generates a lot of nested local declarations and variable copies. A source-to-source compilation pass then simplifies the generated code.

Compiling μF to JAX. From the purely functional μF code, it is now possible to generate Python code that is compatible with JAX. Stream functions are compiled to classes with an `init` and a `step` method. To avoid variable scoping issues in Python, a compilation pass rewrites the μF code without nested definitions. The compilation to Python is then straightforward. The `match` construct is translated into JAX conditionals that take closures as arguments to control their evaluation. Finally, to enable automatic parallelization, data structures must be serializable. Since the state of ProbZelus programs respects a simple tree structure, the `@register_pytree_node_class` decorator can be used to automatically generate the serialization and deserialization functions.

For example, the generated code for the `coin` model in Section 2 is the following:

```
@register_pytree_node_class
class coin(Node):
  def init(self):
    return {"theta": 42.0, "fst": True}
```

$$
\begin{array}{rcl}
C_p(\texttt{skip}) & = & p \\
C_p(x := e) & = & \texttt{let } x = C(e) \texttt{ in } p \\
C_p(\texttt{state}(x) := e) & = & \texttt{let } x = C(e) \texttt{ in } p \\
C_p(\texttt{var } x \texttt{ in } S) & = & \texttt{let } (p, x) = C_{(p,x)}(S) \texttt{ in } p \\
C_p(S_1 \texttt{ ; } S_2) & = & \texttt{let } p = C_p(S_1) \texttt{ in } C_p(S_2) \\
C_p(\texttt{match } e \texttt{ with } \mid C_1 \texttt{ -> } S_1 \ldots \mid C_n \texttt{ -> } S_n) & = & \texttt{match } C(e) \texttt{ with } \mid C_1 \texttt{ -> } C_p(S_1) \ \ldots \mid C_n \texttt{ -> } C_p(S_n) \\
C_p(o.\texttt{reset}) & = & \texttt{let } o = \texttt{init}(m) \texttt{ in } p \qquad \text{if } o : m \\
C_p(o.\texttt{reset}) & = & \texttt{let } o = \texttt{infer}(m) \texttt{ in } p \qquad \text{if } o : \texttt{infer}(m) \\
C_p(p' := o.\texttt{step}(e)) & = & \texttt{let } (p', o) = \texttt{unfold}(o, C(e)) \texttt{ in } p \\
C_p(p' := o.\texttt{pstep}(e)) & = & \texttt{let } ((p', proba), o) = \texttt{unfold}(o, (proba, C(e))) \texttt{ in } p \\
C_p(\texttt{sample}(proba, e)) & = & \texttt{let } (p', proba) = \texttt{sample}(proba, C(e)) \texttt{ in } p \\
C_p(\texttt{observe}(proba, e_1, e_2)) & = & \texttt{let } proba = \texttt{sample}(proba, C(e)) \texttt{ in } p
\end{array}
$$

Figure 6. Compiling OBC to μF.

```
def step(self, *args):
  (state, (proba, obs)) = args
  def _t(_):
    u_dist = uniform_float(0.0, 1.0)
    (proba, theta) = sample(proba, u_dist)
    return (proba, {**state, "theta": theta})
  def _f(_):
    return (proba, state)
  (proba, state) = cond(state["fst"], _t, _f, None)
  state = {**state, "fst": False}
  b_dist = bernoulli(state["theta"]
  (proba, ()) = observe(proba, b_dist, obs)
  return (state, (proba, state["theta"]))
```

5 Evaluation

We evaluate the performance of our new inference engine for ProbZelus on a set of existing benchmarks [2]. The missing benchmarks are models that require more advanced inference algorithms (SLAM and MTT) or linear algebra solvers (Discrete-time Algebraic Riccati Equation) that are not yet implemented in JAX (robot). In our evaluation, we consider two questions: (**QR1**) What is the impact of the new inference engine on accuracy? (**QR2**) Does automatic parallelization improve runtime performance?

5.1 Experimental method

For each of the examples, we measure the execution time and the accuracy obtained after 500 steps with the OCaml inference engine on CPU, and the JAX inference engine on GPU. Accuracy is the *Mean Square Error* (MSE) over all the parameters of the model. For each experiment, we compute the median and the 90% and 10% quantiles, aggregated over 10 runs. All of the experiments were performed on an Intel Xeon E7 server with 64 cores, 128 GB of RAM and an Nvidia Quadro M6000 GPU with 24 GB of RAM.

The selected benchmarks are the following:

Coin estimates the bias of a coin from a series of tosses. We initially assume that the bias follows a uniform distribution over $[0, 1]$. This model corresponds to the `coin` example presented in Section 2.

Gaussian-Gaussian estimates the mean and standard deviation of a normal distribution from a set of samples. We initially assume that the mean follows a distribution $\mathcal{N}(0, 10)$, and that the standard deviation follows a distribution $\mathcal{N}(0, 1)$.

Kalman3D is a 3D version of the `hmm` model of Section 3.4 where an agent estimates its position from noisy observations. At each instant, we assume that the current position follows a normal distribution around the previous position, and that the current observation follows a normal distribution around the current position.

Outlier is a model adapted from [21]. As with Kalman, an agent tries to estimate its position, but the sensor is faulty and occasionally produces invalid readings. The probability of getting an invalid read follows a Beta$(100, 1000)$ distribution, which corresponds to approximately 10% of errors. If a reading is invalid, we assume that the current observation follows an uninformative normal distribution $\mathcal{N}(0, 1000)$. Otherwise the model is the same as Kalman.

5.2 Results

QR1: Accuracy. Accuracy results are presented in Figure 7. For all the examples considered, the two inference engines are equivalent. As expected the error decreases when the number of particles increases before reaching a plateau when the results approach the theoretical distribution. Minor differences are due to implementation differences of OCaml and JAX random generators.

QR2: Runtime Performance. The runtime performance results are presented next to the accuracy results in Figure 8. Performances of the OCaml runtime are characteristic of the classic time/accuracy trade-off: an higher number of particles yields better estimates at the cost of longer execution times. The performance of the JAX inference engine remains nearly constant up to $100,000$ particles. Compiling to GPU optimized code is expensive and only amortized for a high number of particles (around $10,000$). On the other hand, once the code is compiled, JAX can easily run a very large number of particles in parallel with very little additional cost. For all models, GPU limits are reached at around $200,000$ particles.

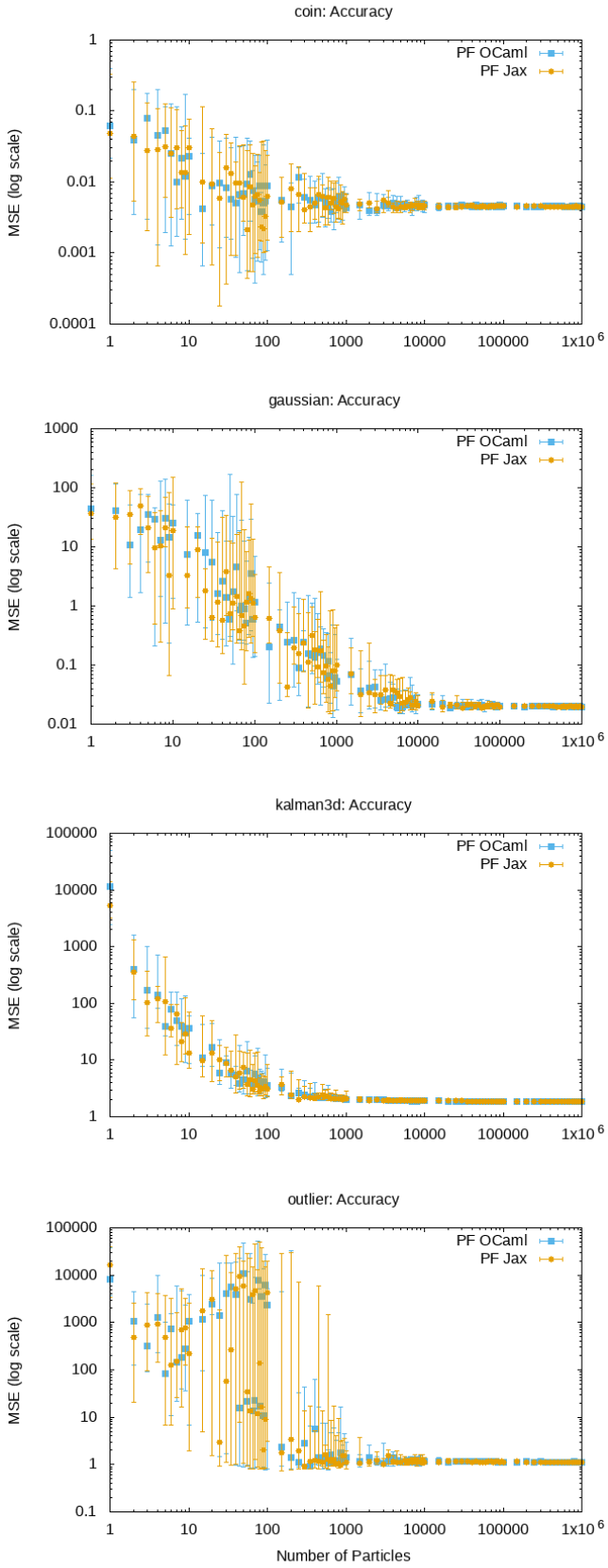


Figure 7. Accuracy as function of number of particles.

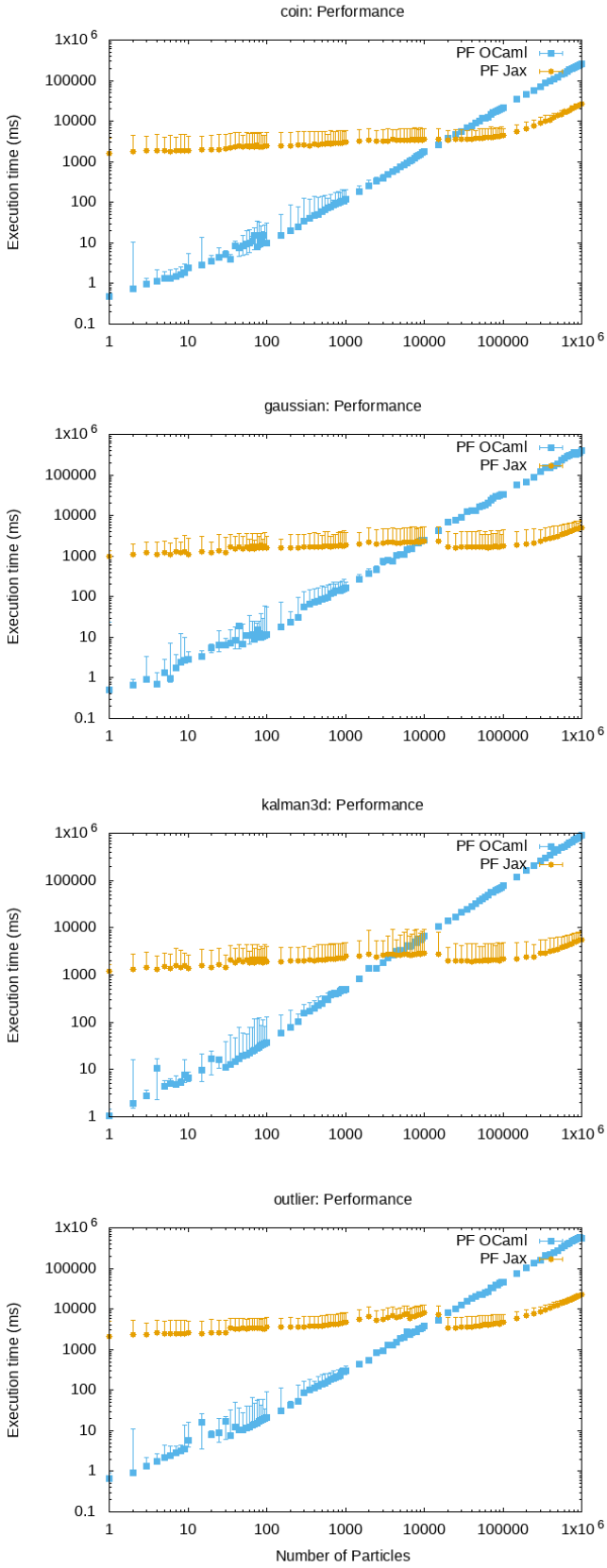


Figure 8. Performance as function of number of particles.

Then performance degrades linearly with the number of particles, but for a given number of particles the JAX runtime still outperforms the OCaml runtime. Note that the crossing point between OCaml and JAX performances depends on the model dimension and complexity (e.g., 20, 000 for Coin, 4, 000 for Kalman3D).

Trade-off. For simple models (Coin, Kalman3D) relatively few particles are required to reach the accuracy plateau. The OCaml implementation thus converges faster than the JAX implementation. However, for more complex models (Gaussian-Gaussian, Outlier), inference converges after the performance crossing point, i.e., the JAX implementation outperforms the OCaml implementation. In addition, the models considered here are all relatively simple with few low dimensional random variables. The difference between JAX and OCaml implementations should be more significant for more realistic models.

6 Related Work and Conclusion

Parallelizing inference algorithms for probabilistic programming is an active research field. Most modern probabilistic languages [7, 15, 24, 26] rely on efficient parallel computing systems popularized by deep learning. Our work, however, is the first to apply these techniques to a reactive language like ProbZelus. There are also several works on the parallelization of synchronous languages [13, 17, 23]. But none of them deal with the specific problems of probabilistic programming.

In this article we showed how JAX can be used to efficiently parallelize the inference engine for reactive probabilistic models programmed in ProbZelus. Using our implementation, we can now try to leverage other JAX features, such as auto-differentiation, to implement more advanced inference techniques for reactive systems involving probabilistic models.

References

[1] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. 2020. Programming Reactive Probabilistic Applications. In *PROBPROG*.

[2] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. 2020. Reactive Probabilistic Programming. In *PLDI*. https://doi.org/10.1145/3385412.3386009

[3] Guillaume Baudart, Louis Mandel, and Reyyan Tekin. 2022. *Reproduction package for article JAX Based Parallel Inference for Reactive Probabilistic Programming*. https://doi.org/10.1145/3462319

[4] Albert Benveniste, Timothy Bourke, Benoît Caillaud, Bruno Pagano, and Marc Pouzet. 2014. A type-based analysis of causality loops in hybrid systems modelers. In *HSCC*. https://doi.org/10.1145/2562059.2562125

[5] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (2003), 64–83. https://doi.org/10.1109/JPROC.2002.805826

[6] Dariusz Biernacki, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. 2008. Clock-directed modular code generation for synchronous data-flow languages. In *LCTES*. https://doi.org/10.1145/1375657.1375674

[7] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* 20 (2019), 28:1–28:6.

[8] Timothy Bourke, Lélio Brun, and Marc Pouzet. 2020. Mechanized semantics and verified compilation for a dataflow synchronous language with reset. In *POPL*. https://doi.org/10.1145/3371112

[9] Timothy Bourke and Marc Pouzet. 2013. *Zelus, a Hybrid Synchronous Language*. École normale supérieure. https://zelus.di.ens.fr.

[10] Timothy Bourke and Marc Pouzet. 2013. Zélus: a synchronous language with ODEs. In *HSCC*. https://doi.org/10.1145/2461328.2461348

[11] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. JAX: composable transformations of Python+NumPy programs. https://github.com/google/jax.

[12] Paul Caspi and Marc Pouzet. 1998. A Co-iterative Characterization of Synchronous Stream Functions. In *CMCS*. https://doi.org/10.1016/S1571-0661(04)00050-7

[13] Albert Cohen, Léonard Gérard, and Marc Pouzet. 2012. Programming parallelism with futures in Lustre. In *EMSOFT*. https://doi.org/10.1145/2380356.2380394

[14] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. 2017. SCADE 6: A formal language for embedded critical software development. In *TASE*. https://doi.org/10.1109/TASE.2017.8285623

[15] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: a general-purpose probabilistic programming system with programmable inference. In *PLDI*. https://doi.org/10.1145/3314221.3314642

[16] Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. 2006. Sequential Monte Carlo samplers. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 68, 3 (2006), 411–436.

[17] Alain Girault. 2005. A Survey of Automatic Distribution Method for Synchronous Programs. In *SLAP*.

[18] Noah D. Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org.

[19] Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet. 2012. A modular memory optimization for synchronous data-flow languages: application to arrays in a lustre compiler. In *LCTES*. https://doi.org/10.1145/2345141.2248426

[20] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The Synchronous Dataflow Programming Language Lustre. *Proc. IEEE* 79, 9 (September 1991), 1305–1320.

[21] Thomas P. Minka. 2001. Expectation Propagation for approximate Bayesian inference. In *UAI*.

[22] Lawrence M. Murray and Thomas B. Schön. 2018. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control* 46 (2018), 29–43.

[23] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. 2011. Multi-task Implementation of Multi-periodic Synchronous Programs. *Discrete Event Dynamic Systems* 21, 3 (2011), 307–338. https://doi.org/10.1007/s10626-011-0107-x

[24] Du Phan, Neeraj Pradhan, and Martin Jankowiak. 2019. Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro. *arXiv:1912.11554* (2019). https://doi.org/10.48550/arXiv.1912.11554

[25] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank D. Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. In *IFL*. https://doi.org/10.1145/3064899.3064910

[26] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep Probabilistic Programming. In *ICLR*.