Etalumis: Bringing Probabilistic Programming to Scientific Simulators at Scale

Atılım Güneş Baydin
University of Oxford

Lei Shao
Intel Corporation

Wahid Bhimji
Lawrence Berkeley National Laboratory

Lukas Heinrich
CERN

Lawrence Meadows
Intel Corporation

Jialin Liu
Lawrence Berkeley National Laboratory

Andreas Munk
University of British Columbia

Saeid Naderiparizi
University of British Columbia

Bradley Gram-Hansen
University of Oxford

Gilles Louppe
University of Liège

Mingfei Ma
Intel Corporation

Xiaohui Zhao
Intel Corporation

Philip Torr
University of Oxford

Victor Lee
Intel Corporation

Kyle Cranmer
New York University

Prabhat
Lawrence Berkeley National Laboratory

Frank Wood
University of British Columbia

ABSTRACT

Probabilistic programming languages (PPLs) are receiving widespread attention for performing Bayesian inference in complex generative models. However, applications to science remain limited because of the impracticability of rewriting complex scientific simulators in a PPL, the computational cost of inference, and the lack of scalable implementations. To address these, we present a novel PPL framework that couples directly to existing scientific simulators through a cross-platform probabilistic execution protocol and provides Markov chain Monte Carlo (MCMC) and deep-learning-based inference compilation (IC) engines for tractable inference. To guide IC inference, we perform distributed training of a dynamic 3DCNN–LSTM architecture with a PyTorch-MPI-based framework on 1,024 32-core CPU nodes of the Cori supercomputer with a global mini-batch size of 128k: achieving a performance of 450 Tflop/s through enhancements to PyTorch. We demonstrate a Large Hadron Collider (LHC) use-case with the C++ Sherpa simulator and achieve the largest-scale posterior inference in a Turing-complete PPL.

1 INTRODUCTION

Probabilistic programming [71] is an emerging paradigm within machine learning that uses general-purpose programming languages to express probabilistic models. This is achieved by introducing statistical conditioning as a language construct so that inverse problems can be expressed. Probabilistic programming languages (PPLs) have semantics [67] that can be understood as Bayesian inference [13, 24, 26]. The major challenge in designing useful PPL systems is that language evaluators must solve arbitrary, user-provided inverse problems, which usually requires general-purpose inference algorithms that are computationally expensive.

In this paper we report our work that enables, for the first time, the use of *existing* stochastic simulator code as a probabilistic program in which one can do fast, repeated (amortized) Bayesian inference; this enables one to predict the distribution of input parameters and all random choices in the simulator from an observation of its output. In other words, given a simulator of a generative process in the forward direction (inputs→outputs), our technique can provide the reverse (outputs→inputs) by predicting the whole latent state of the simulator that could have given rise to an observed instance of its output. For example, using a particle physics simulation we can get distributions over the particle properties and decays within the simulator that can give rise to a collision event observed in a detector, or, using a spectroscopy simulator we can determine the elemental matter composition and dispersions within the simulator explaining an observed spectrum. In fields where accurate simulators of real-world phenomena exist, our technique enables the interpretable explanation of real observations under the structured model defined by the simulator code base.

We achieve this by defining a probabilistic programming execution protocol that interfaces with existing simulators at the sites of random number draws, without altering the simulator's structure and execution in the host system. The random number draws are routed through the protocol to a PPL system which treats these as samples from corresponding prior distributions in a Bayesian setting, giving one the capability to record or guide the execution of the simulator to perform inference. Thus we generalize existing simulators as probabilistic programs and make them subject to inference under general-purpose inference engines.

Inference in the probabilistic programming setting is performed by sampling in the space of execution traces, where a single sample (an execution trace) represents a full run of the simulator. Each

execution trace itself is composed of a potentially unbounded sequence of addresses, prior distributions, and sampled values, where an address is a unique label identifying each random number draw. In other words, we work with empirical distributions over simulator executions, which entails unique requirements on memory, storage, and computation that we address in our implementation. The addresses comprising each trace give our technique the unique ability to provide direct connections to the simulator code base for any predictions at test time, where the simulator is no longer used as a black box but as a highly structured and interpretable probabilistic generative model that it implicitly represents.

Our PPL provides inference engines from the Markov chain Monte Carlo (MCMC) and importance sampling (IS) families. MCMC inference guarantees closely approximating the true posterior of the simulator, albeit with significant computational cost due to its sequential nature and the large number of iterations one needs to accumulate statistically independent samples. Inference compilation (IC) [47] addresses this by training a dynamic neural network to provide proposals for IS, leading to fast amortized inference.

We name this project "Etalumis", the word "simulate" spelled backwards, as a reference to the fact that our technique essentially inverts a simulator by probabilistically inferring all choices in the simulator given an observation of its output. We demonstrate this by inferring properties of particles produced at the Large Hadron Collider (LHC) using the Sherpa[1] [29] simulator.

1.1 Contributions

Our main contributions are:

- A novel PPL framework that enables execution of existing stochastic simulators under the control of general-purpose inference engines, with HPC features including handling multi-TB data and distributed training and inference.
- The largest scale posterior inference in a Turing-complete PPL, where our experiments encountered approximately 25,000 latent variables[2] expressed by the existing Sherpa simulator code base of nearly one million lines of code in C++ [29].
- Synchronous data parallel training of a dynamic 3DCNN–LSTM neural network (NN) architecture using the PyTorch [61] MPI framework at the scale of 1,024 nodes (32,768 CPU cores) with a global minibatch size of 128k. To our knowledge this is the largest scale use of PyTorch's builtin MPI functionality,[3] and the largest minibatch size used for this form of NN model.

2 PROBABILISTIC PROGRAMMING FOR PARTICLE PHYSICS

Particle physics seeks to understand particles produced in collisions at accelerators such at the LHC at CERN. Collisions happen millions of times per second, creating cascading particle decays, observed in complex instruments such as the ATLAS detector [2], comprising millions of electronics channels. These experiments analyze the vast volume of resulting data and seek to reconstruct the initial particles produced in order to make discoveries including physics beyond the current Standard Model of particle physics [28][73][63][72].

The Standard Model has a number of parameters (e.g., particle masses), which we can denote θ, describing the way particles and fundamental forces act in the universe. In a given collision at the LHC, with initial conditions denoted E, we observe a cascade of particles interact with particle detectors. If we denote *all* of the random "choices" made by nature as $\mathbf{x}$, the Standard Model describes, generatively, the conditional probability $p(\mathbf{x}|E,\theta)$, that is, the distribution of all choices $\mathbf{x}$ as a function of initial conditions E and model parameters θ. Note that, while the Standard Model can be expressed symbolically in mathematical notation [32, 62], it can also be expressed computationally as a stochastic simulator [29], which, given access to a random number generator, can draw samples from $p(\mathbf{x})$.[4] Similarly, a particle detector can be modeled as a stochastic simulator, generating samples from $p(\mathbf{y}|\mathbf{x})$, the likelihood of observation $\mathbf{y}$ as a function of $\mathbf{x}$.

In this paper we focus on a real use-case in particle physics, performing experiments on the decay of the τ (tau) lepton. This is under active investigation by LHC physicists [4] and important to uncovering properties of the Higgs boson. We use the state-of-the-art Sherpa simulator [29] for modeling τ particle creation in LHC collisions and their subsequent decay into further particles (the stochastic events $\mathbf{x}$ above), coupled to a fast 3D detector simulator for the detector observation $\mathbf{y}$.

Current methods in the field include performing classification and regression using machine learning approaches on low dimensional distributions of derived variables [4] that provide point-estimates without the posterior of the full latent state nor the deep interpretability of our approach. Inference of the latent structure has only previously been used in the field with drastically simplified models of the process and detector [43] [3].

PPLs allow us to express inference problems such as: given an actual particle detector observation $\mathbf{y}$, what sequence of choices $\mathbf{x}$ are likely to have led to this observation? In other words, we would like to find $p(\mathbf{x}|\mathbf{y})$, the distribution of $\mathbf{x}$ as a function of $\mathbf{y}$. To solve this inverse problem via conditioning requires invoking Bayes rule

$$p(\mathbf{x}|\mathbf{y}) = \frac{p(\mathbf{y},\mathbf{x})}{p(\mathbf{y})} = \frac{p(\mathbf{y}|\mathbf{x})p(\mathbf{x})}{\int p(\mathbf{y}|\mathbf{x})p(\mathbf{x})d\mathbf{x}}$$

where the posterior distribution of interest, $p(\mathbf{x}|\mathbf{y})$, is related to the composition of the two stochastic simulators in the form of the joint distribution $p(\mathbf{y},\mathbf{x}) = p(\mathbf{y}|\mathbf{x})p(\mathbf{x})$ renormalized by the marginal probability, or evidence of the data, $p(\mathbf{y}) = \int p(\mathbf{y}|\mathbf{x})p(\mathbf{x})d\mathbf{x}$. Computing the evidence requires summing over all possible paths that the simulation can take. This is a large number of possible paths; in most models this is a quantity that is impossible to compute in polynomial time. In practice PPLs approximate the posterior $p(\mathbf{x}|\mathbf{y})$ using sampling-based inference engines that sidestep the integration problem but remain computationally intensive. This specifically is where probabilistic programming meets, for the first time in this paper, high-performance computing.

[1] https://gitlab.com/sherpa-team/sherpa

[2] Note that the simulator defines an unlimited number of random variables because of the presence of rejection sampling loops.

[3] Personal communication with PyTorch developers.

[4] Dropping the dependence on E and θ because everything in this example is conditionally dependent on these quantities.

3 STATE OF THE ART

3.1 Probabilistic programming

Within probabilistic programming, recent advances in computational hardware have made it possible to distribute certain types of inference processes, enabling inference to be applied to problems of real-world relevance [70]. By parallelizing computation over several cores, PPLs have been able to perform large-scale inference on models with increasing numbers of observations, such as the cause and effect analysis of 1.6×10^9 genetic measurements [30, 70], spatial analysis of 1.5×10^4 shots from 308 NBA players [18], exploratory analysis of 1.7×10^6 taxi trajectories [36], and probabilistic modeling for processing hundreds-of-thousands of Xbox live games per day to rank and match players fairly [33, 55].

In all these large-scale programs, despite the number of observations being large, model sizes in terms of the number of latent variables have been limited [36]. In contrast, to perform inference in a complex scientific model such as the Standard Model encoded by Sherpa requires handling thousands of latent variables, all of which need to be controlled within the program to perform inference in a scalable manner. To our knowledge, no existing PPL system has been used to run inference at the scale we are reporting in this work, and instances of distributed inference in existing literature have been typically restricted to small clusters [19].

A key feature of PPLs is that they decouple model specification from inference. A model is implemented by the user as a stand-alone regular program in the host programming language, specifying a generative process that produces samples from the joint prior distribution $p(\mathbf{y}, \mathbf{x}) = p(\mathbf{y}|\mathbf{x})p(\mathbf{x})$ in each execution, that is, a forward model going from choices $\mathbf{x}$ to outcomes (observations) $\mathbf{y}$. The same program can then be executed using a variety of general-purpose inference engines available in the PPL system to obtain $p(\mathbf{x}|\mathbf{y})$, the inverse going from observations $\mathbf{y}$ to choices $\mathbf{x}$. Inference engines available in PPLs range from MCMC-based lightweight Metropolis Hastings (LMH) [74] and random-walk Metropolis Hastings (RMH) [46] algorithms to importance sampling (IS) [8] and sequential Monte Carlo [22]. Modern PPLs such as Pyro [11] and TensorFlow Probability [19, 70] use gradient-based inference engines including variational inference [36, 42] and Hamiltonian Monte Carlo [37, 57] that benefit from modern deep learning hardware and automatic differentiation [9] features provided by PyTorch [61] and TensorFlow [5] libraries. Another way of making use of gradient-based optimization is to combine IS with deep-learning-based proposals trained with data sampled from the probabilistic program, resulting in the IC algorithm [47, 49] in an amortized inference setting [25].

3.2 Distributed training for deep learning

To perform IC inference in Turing-complete PPLs in general, we would like to support the training of dynamic NNs whose runtime structure changes in each execution of the probabilistic model by rearranging NN modules corresponding to different addresses (unique random number draws) encountered [47] (Section 4.3). Moreover, depending on probabilistic model complexity, the NNs may grow in size if trained in an online setting, as a model can represent a potentially unbounded number of random number draws. In addition to these, the volume of training data required is large, as the data keeps track of all execution paths within the simulator. To

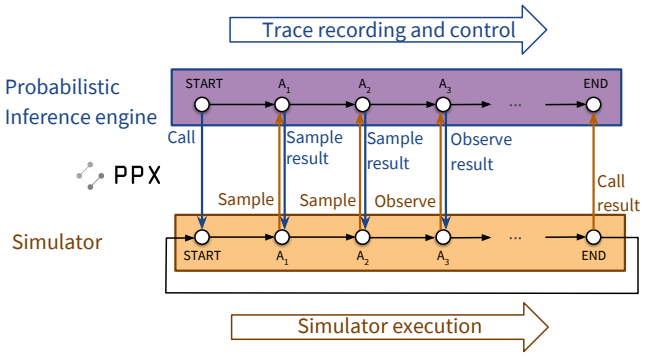

Figure 1: The probabilistic execution protocol (PPX). *Sample* and *observe* statements correspond to random number draws and conditioning, respectively.

enable rapid prototyping, model evaluation, and making use of HPC capacity, scaling deep learning training to multiple computation units is highly desirable [38, 44, 45, 51, 52].

In this context there are three prominent parallelism strategies: data- and model-parallelism, and layer pipelining. In this project we work in a data-parallel setting where different nodes train the same model on different subsets of data. For such training, there are synchronous- and asynchronous-update approaches. In synchronous update [16, 59], locally computed gradients are summed across the nodes at the same time with synchronization barriers for parameter update. In asynchronous update [17, 58, 77], one removes the barrier so that nodes can independently contribute to a parameter server. Although synchronous update can entail challenges due to straggler effects [15, 69], it has desirable properties in terms of convergence, reproducibility, and ease of debugging. In this work, given the novelty of the probabilistic techniques we are introducing and the need to fully understand and compare trained NNs without ambiguity, we employ synchronous updates.

In synchronous updates, large global minibatches can make convergence challenging and hinder test accuracy. Keskar et al. [39] pointed out large-minibatch training can lead to sharp minima and a generalization gap. Other work [31, 76] argues that the difficulties in large-minibatch training are optimization related and can be mitigated with learning rate scaling [31]. You et al. [76] apply layer-wise adaptive rate scaling (LARS) to achieve large-minibatch-size training of a Resnet-50 architecture without loss of accuracy, and Ginsburg et al. [27] use layer-wise adaptive rate control (LARC) to improve training stability and speed. Smith et al. [65] have proposed to increase the minibatch size instead of decaying the learning rate, and more recent work [53, 64] showed relationships between gradient noise scale (or training steps) and minibatch size. Through such methods, distributed training has been scaled to many thousands of CPUs or GPUs [44, 45, 51, 54]. While we take inspiration from these recent approaches, our dynamic NN architecture and training data create a distinct training setting which requires appropriate innovations, as discussed in Section 4.3.

4 INNOVATIONS

4.1 PPX and pyprob: executing existing simulators as probabilistic programs

One of our main contributions in Etalumis is the development of a probabilistic programming execution protocol (PPX), which defines a cross-platform API for the execution and control of stochastic simulators[5] (Figure 1). The protocol provides language-agnostic definitions of common probability distributions and message pairs covering the call and return values of: (1) program entry points; (2) *sample* statements for random number draws; and (3) *observe* statements for conditioning. The purpose of this protocol is twofold:

- It allows us to record execution traces of a stochastic simulator as a sequence of *sample* and *observe* (conditioning) operations on random numbers, each associated with an address A_t. We can use these traces for tasks such as inspecting the probabilistic model implemented by the simulator, computing likelihoods, learning surrogate models, and generating training data for IC NNs.
- It allows us to control the execution of the simulator, at inference time, by making intelligent choices for each random number draw as the simulator keeps requesting random numbers. General-purpose PPL inference guides the simulator by making random number draws not from the prior $p(\mathbf{x})$ but from proposal distributions $q(\mathbf{x}|\mathbf{y})$ that depend on observed data $\mathbf{y}$ (Section 2).

PPX is based on flatbuffers,[6] a streamlined version of Google protocol buffers, providing bindings into C++, C#, Go, Java, JavaScript, PHP, Python, and TypeScript, enabling lightweight PPL front ends in these languages—in the sense of requiring the implementation of a simple intermediate layer to perform *sample* and *observe* operations over the protocol. We exchange PPX messages over ZeroMQ[7] [34] sockets, which allow communication between separate processes in the same machine (via inter-process sockets) or across a network (via TCP). PPX is inspired by the Open Neural Network Exchange (ONNX) project[8] allowing interoperability between major deep learning frameworks, and it allows the execution of any stochastic simulator under the control of any PPL system, provided that the necessary bindings are incorporated on both sides.

Using the PPX protocol as the interface, we implement two main components: (1) pyprob, a PyTorch-based PPL[9] in Python and (2) a C++ binding to the protocol to route the random number draws in Sherpa to the PPL and therefore allow probabilistic inference in this simulator. Our PPL is designed to work with models written in Python and other languages supported through PPX. This is in contrast to existing PPLs such as Pyro [11] and TensorFlow Probability [19, 70] which do not provide a way to interface with existing simulators and require one to implement any model from scratch in the specific PPL.[10] We develop pyprob based on PyTorch [61], to utilize its automatic differentiation [9] infrastructure with support for dynamic computation graphs for IC inference.

[5] https://github.com/probprog/ppx
[6] http://google.github.io/flatbuffers/
[7] http://zeromq.org/
[8] https://onnx.ai/
[9] https://github.com/probprog/pyprob
[10] We are planning to provide PPX bindings for these PPLs in future work.

4.2 Efficient Bayesian inference

Working with existing simulators as probabilistic programs restricts the class of inference engines that we can put to use. Modern PPLs commonly use gradient-based inference such as Hamiltonian Monte Carlo [57] and variational inference [36, 42] to approximate posterior distributions. However this is not applicable in our setting due to the absence of derivatives in general simulator codes. Therefore in pyprob we focus our attention on two inference engine families that can control Turing-complete simulators over the PPX protocol: MCMC in the RMH variety [46, 74], which provides a high-compute-cost sequential algorithm with statistical guarantees to closely approximate the posterior, and IS with IC [47], which does not require derivatives of the simulator code but still benefits from gradient-based methods by training proposal NNs and using these to significantly speed up IS inference.

It is important to note that the inference engines in pyprob work in the space of execution traces of probabilistic programs, such that a single sample from the inference engine corresponds to a full run of the simulator. Inference in this setting amounts to making adjustments to the random number draws, re-executing the simulator, and scoring the resulting execution in terms of the likelihood of the given observation. Depending on the specific observation and the simulator code involved, inference is computationally very expensive, requiring up to millions of executions in the RMH engine. Despite being very costly, RMH provides a way of sampling from the true posterior [56, 57], which is needed in initial explorations of any new simulator to establish correct posteriors serving as reference to confirm that IC inference can work correctly in the given setting. To establish the correctness of our inference results, we implement several MCMC convergence diagnostics. Autocorrelation measures the number of iterations one needs to get effectively independent samples in the same MCMC chain, which allows us to estimate how long RMH needs to run to reach a target effective sample size. The Gelman–Rubin metric, given multiple independent MCMC chains sampled from the same posterior, compares the variance of each chain to the pooled variance of all chains to statistically establish that we converged on the true posterior [24].

RMH comes with a high computational cost. This is because it requires a large number of initial samples to be generated that are then discarded, of the order $\sim 10^6$ for the Sherpa model we present in this paper. This is required to find the posterior density, which, as the model begins from an arbitrary point of the prior, can be very far from the starting region. Once this "burn-in" stage is completed the MCMC chain should be sampling from within the region containing the posterior. In addition to this, the sequential nature of each chain limits our ability to parallelize the computation, again creating computational inefficiencies in the high-dimensional space of simulator execution traces that we work with in our technique.

In order to provide fast, repeated inference in a distributed setting, we implement the IC algorithm, which trains a deep recurrent NN to provide proposals for an IS scheme [47]. This works by running the simulator many times and therefore sampling a large set of execution traces from the simulator prior $p(\mathbf{x}, \mathbf{y})$, and using these to train a NN that represents $q(\mathbf{x}|\mathbf{y})$, i.e., informed proposals for random number draws $\mathbf{x}$ given observations $\mathbf{y}$, by optimizing the loss $\mathcal{L}(\phi) = \mathbb{E}_{p(\mathbf{y})}\left[D_{\mathrm{KL}}(p(\mathbf{x}|\mathbf{y})||q_\phi(\mathbf{x}|\mathbf{y}))\right] = \mathbb{E}_{p(\mathbf{x},\mathbf{y})}\left[-\log q_\phi(\mathbf{x}|\mathbf{y})\right] +$

const., where ϕ are NN parameters (Algorithm 1 and Figure 3) [47]. This phase of sampling the training data and training the NN is costly, but it needs to be performed only once for any given model. Once the proposal NN is trained to convergence, the IC inference engine becomes competitive in performance, which allows us to achieve a given effective sample size in the posterior $p(\mathbf{x}|\mathbf{y})$ using a fraction of the RMH computational cost. IC inference is embarrassingly parallel, where many instances of the same trained NN can be executed to run distributed inference on a given observation.

To further improve inference performance, we make several low-level improvements in the code base. The C++ front end of PPX uses concatenated stack frames of each random number draw as a unique address identifying a latent variable in the corresponding PPL model. Stack traces are obtained with the `backtrace(3)` function as instruction addresses and then converted to symbolic names using the `dladdr(3)` function [50]. The conversion is quite expensive, which prompted us to add a hash map to cache `dladdr` results, giving a 5x improvement in the production of address strings that are essential in our inference engines. The particle detector simulator that we use was initially coded to use the xtensor library[11] to implement the probability density function (PDF) of multivariate normal distributions in the general case, but was exclusively called on 3D data. This code was replaced by a scalar-based implementation limited to the 3D case, resulting in a 13x speed-up in the PDF, and a 1.5x speed-up of our simulator pipeline in general. The bulk of our further optimizations focus on the NN training for IC inference and are discussed in the next sections.

4.3 Dynamic neural network architecture

The NN architecture used in IC inference is based on a LSTM [35] recurrent core that gets executed as many time steps as the simulator's probabilistic trace length (Figure 3). To this core NN, various other NN components get attached according to the series of addresses A_t executed in the simulator. In other words, we construct a dynamic NN whose runtime structure changes in each execution trace, implemented using the dynamic computation graph infrastructure in PyTorch. The input to this LSTM in each time step is a concatenation of embeddings of the observation, the current address in the simulator, and the previously sampled value. The observation embedding is a NN specific to the observation domain. Address embeddings are learned vectors representing the identity of random choices A_t in the simulator address space. Sample embeddings are address-specific layers encoding the value of the random draw in the previous time step. The LSTM output, at each time step, is fed into address-specific proposal layers that provide the final output of the NN for IC inference: proposal distributions $q(\mathbf{x}|\mathbf{y})$ to use for each address A_t as the simulator keeps running and requesting new random numbers over the PPX protocol (Section 4.1).

For the Sherpa experiments reported in this paper, we work with 3D observations of size 35x35x20, representing particle detector voxels. To tune NN architecture hyperparameters, we search a grid of LSTM stacks in range {1, 4}, LSTM hidden units in the set {128, 256, 512}, and number of proposal mixture components in the set {5, 10, 25, 50} (Figure 2). We settle on the following architecture: an LSTM with 512 hidden units; an observation embedding of size 256,

[11] https://xtensor.readthedocs.io

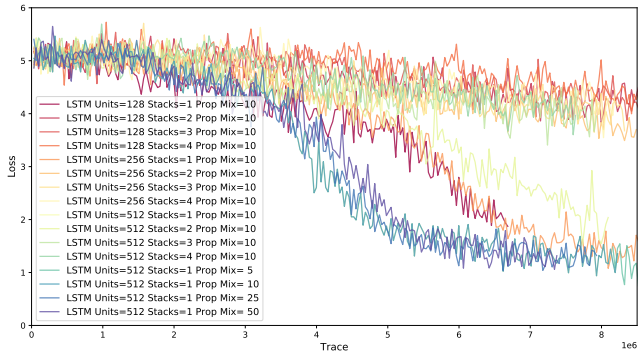

Figure 2: Loss curves for NN architectures considered in the hyperparameter search detailed in the text.

encoded with a 3D convolutional neural network (CNN) [48] acting as a feature extractor, with layer configuration Conv3D(1, 64, 3)–Conv3D(64, 64, 3)–MaxPool3D(2)–Conv3D(64, 128, 3)–Conv3D(128, 128, 3)–Conv3D(128, 128, 3)– MaxPool3D(2)–FC(2048, 256); previous sample embeddings of size 4 given by single-layer NNs; and address embeddings of size 64. The proposal layers are two-layer NNs, the output of which are either a mixture of ten truncated normal distributions [12] (for uniform continuous priors) or a categorical distribution (for categorical priors). We use ReLU nonlinearities in all NN components. All of these NN components except the LSTM and the 3DCNN are dependent on addresses A_t in the simulator, and these address-specific layers are created at the first encounter with a random number draw at a given address. Thus the number of trainable parameters in an IC NN is dependent on the size of the training data, because the more data gets used, the more likely it becomes to encounter new addresses in the simulator.

The pyprob framework is capable of operating in an "online" fashion, where NN training and layer generation happens using traces sampled by executing the simulator on-the-fly and discarding traces after each minibatch, or "offline", where traces are sampled from the simulator and saved to disk as a dataset for further reuse (Algorithm 2). In our experiments, we used training datasets of 3M and 15M traces, resulting in NN sizes of 156,960,440 and 171,732,688 parameters respectively. All timing and scaling results presented in Sections 6.1 and 6.2 are performed with the larger network.

4.4 Training of dynamic neural networks

Scalable training of dynamic NNs we introduced in Section 4.3 pose unique challenges. Because of the address-dependent nature of the embedding and proposal layers of the overall IC NN, different nodes/ranks in a distributed training setting will work with different NN configurations according to the minibatch of training data they process at any given time. When the same NN is not shared across all nodes/ranks, it is not possible to rely on a generic allreduce operation for gradient averaging which is required for multi-node synchronous SGD. Inspired by neural machine translation (NMT) [75], in the offline training mode with training data saved on the disk, we implemented the option of pre-processing the whole dataset to pre-generate all embedding and proposal layers that a given dataset would imply to exist. Once layer pre-generation

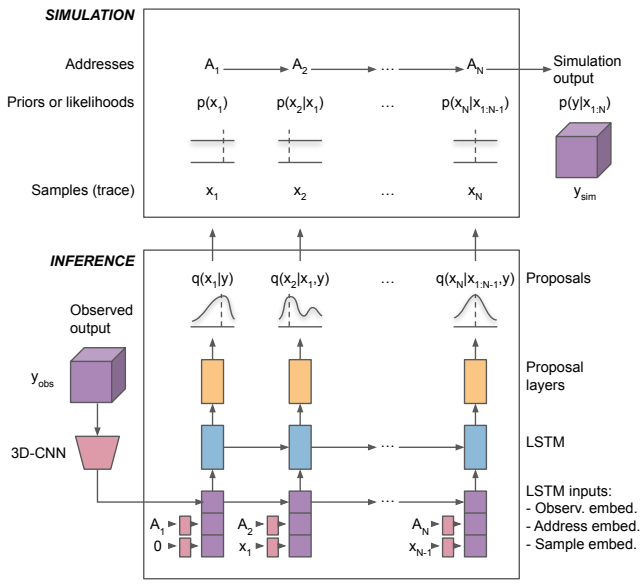

Figure 3: Simulation and inference. *Top:* model addresses, priors and samples. *Bottom:* IC inference engine proposals and NN architecture.

is done, the collection of all embedding and proposal layers are shared on each node/rank. In this way, for offline training, we have a globally shared NN representing the superset of all NN components each node needs to handle in any given minibatch, thus making it possible to scale training of the NN on multiple nodes.

Our allreduce-based training algorithm can also work in the online training setting, where training data is sampled from the simulator on-the-fly, if we freeze a globally shared NN and discard any subsequently encountered traces that contain addresses unknown at the time of NN architecture freezing. In future work, we intend to add a distributed open-ended implementation for online training to allow running without discarding, that will require the NN instances in each node/rank to grow with newly seen addresses.

4.4.1 Single node improvements to Etalumis. We profiled the Etalumis architecture with vtune, Cprofiler, and PyTorch autograd profiler, identifying data loading and 3D convolution as the primary computational hot-spots on which we focused our optimization efforts. We provide details on data loading and 3D convolution in the subsequent sections. In addition to these, execution traces from the Sherpa simulator have many different trace types (a unique sequence of addresses A_t, with different sampled values) with different rates of occurrence: in a given dataset, some trace types can be encountered thousands of times while others are seen only once. This is problematic because at training time we further divide each minibatch into "sub-minibatches" based on trace type, where each sub-minibatch can be processed by the NN in a single forward execution due to all traces being of the same type, i.e., sharing the same sequence of addresses A_t and therefore requiring the same NN structure (Algorithm 1). Therefore minibatches containing more than one trace type do not allow for effective parallelization and vectorization. In other words, unlike conventional NNs, the effective

Algorithm 1 Computing minibatch loss $\mathcal{L}_n$ of NN parameters ϕ

Require: Minibatch $\mathcal{D}_n$
  $L \leftarrow$ number of unique trace types found in $\mathcal{D}_n$
  Construct sub-minibatches $\mathcal{D}_n^l$, for $l = 1, \ldots, L$
  $\mathcal{L}_n \leftarrow 0$
  for $l \in \{1, \ldots, L\}$ **do**
    $\mathcal{L}_n \leftarrow \mathcal{L}_n - \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{D}_n^l} \log q_\phi(\boldsymbol{x}|\boldsymbol{y})$
  end for
  return $\mathcal{L}_n$

Algorithm 2 Distributed training with MPI backend. $p(\boldsymbol{x},\boldsymbol{y})$ is the simulator and $\hat{G}(\boldsymbol{x},\boldsymbol{y})$ is an offline dataset sampled from $p(\boldsymbol{x},\boldsymbol{y})$

Require: OnlineData {True/False value}
Require: B {Minibatch size}
  Initialize inference network $q_\phi(\boldsymbol{x}|\boldsymbol{y})$
  $N \leftarrow$ number of processes
  for all $n \in \{1, \ldots, N\}$ **do**
    while Not Stop **do**
      if OnlineData **then**
        Sample $\mathcal{D}_n = \{(\boldsymbol{x},\boldsymbol{y})_1, \ldots, (\boldsymbol{x},\boldsymbol{y})_B\}$ from $p(\boldsymbol{x},\boldsymbol{y})$
      else
        Get $\mathcal{D}_n = \{(\boldsymbol{x},\boldsymbol{y})_1, \ldots, (\boldsymbol{x},\boldsymbol{y})_B\}$ from $\hat{G}(\boldsymbol{x},\boldsymbol{y})$
      end if
      Synchronize parameters (ϕ) across all processes
      $\mathcal{L}_n \leftarrow -\frac{1}{B}\sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{D}_n} \log q_\phi(\boldsymbol{x}|\boldsymbol{y})$
      Calculate $\nabla_\phi \mathcal{L}_n$
      Call `all_reduce` s.t. $\nabla_\phi \mathcal{L} \leftarrow \frac{1}{N}\sum_{n=1}^{N} \nabla_\phi \mathcal{L}_n$
      Update ϕ using $\nabla_\phi \mathcal{L}$ with e.g. ADAM, SGD, LARC, etc.
    end while
  end for

minibatch size is determined by the average size of sub-minibatches, and the more trace types we have within a minibatch, the slower the computation. To address this, we explored multiple methods to enlarge effective minibatch size, such as sorting traces, multi-bucketing, and selectively batching traces from the same trace type together in each minibatch. These options and their trade offs are described in more detail in Section 7.

4.4.2 Single node improvements to PyTorch. The flexibility of dynamic computation graphs and competitive speed of PyTorch have been crucial for this project. Optimizations were performed on code belonging to Pytorch stable release v1.0.0 to better support this project on Intel® Xeon® CPU platforms, focused in particular on 3D convolution operations making use of the MKL-DNN open source math library. MKL-DNN uses a direct convolution algorithm and for a 5-dimensional input tensor with layout {N, C, D, H, W}, it is reordered into a layout of {N, C, D, H, W, 8c} which is more amenable for SIMD vectorization.[12] The 3D convolution operator is vectorized on the innermost dimension which matches the 256-bit instruction length on AVX2, and parallelized on the outer dimensions. We also performed cache optimization to further improve performance. With these improvements we found the heavily used 3D convolution kernel achieved an 8x improvement on the Cori HSW platform.[13] The overall improvement on single node training

[12] https://intel.github.io/mkl-dnn/understanding_memory_formats.html
[13] https://docs.nersc.gov/analytics/machinelearning/benchmarks/

time is given in Section 6.1. These improvements are made available in a fork of the official PyTorch repository.[14]

4.4.3 I/O optimization. I/O is challenging in many deep learning workloads partly due to random access patterns, such as those induced by shuffling, disturbing any pre-determined access order. In order to reduce the number of random access I/O operations, we developed a parallel trace sorting algorithm and pre-sorted the 15M traces according to trace type (Section 4.4.1). We further grouped the small trace files into larger files, going from 750 files with 20k traces per file to 150 files with 100k traces per file. With this grouping and sorting, we ensured that I/O requests follow a sequential access onto a contiguous file region which further improved the I/O performance. Metadata operations are also costly, so we enhanced the Python shelve module's file open/close performance with a caching mechanism, which allows concurrent access from different ranks to the same file.

Specific to our PPL setting, training data consists of execution traces that have a complex hierarchy, with each trace file containing many trace objects that consist of variable sequences of sample objects representing random number draws, which further contain variable length tensors, strings, integers, booleans, and other generic Python objects. PyTorch serialization with pickle is used to handle the complex trace data structure, but the pickle and unpickle overhead are very high. We developed a "pruning" function to shrink the data by removing non-necessary structures. We also designed a dictionary of simulator addresses A_t, which accumulates the fairly long address strings and assigns shorthand IDs that are used in serialization. This brought a 40% memory consumption reduction as well as large disk space saving.

For distributed training, we developed distributed minibatch sampler and dataset classes conforming to the PyTorch training API. The sampler first splits the sorted trace indices into minibatch-sized chunks, so that all traces in each minibatch are highly likely to be of the same type, then optionally groups these chunks into several buckets (Section 7.2). Within each bucket, the chunks are assigned with a round-robin algorithm to different ranks, such that each rank has roughly same distribution of workload. The distributed sampler enables us to scale the training at 1,024 nodes.

The sorting of traces and their grouping into minibatch chunks significantly improves the training speed (up to 50× in our experiments) by enabling all traces in a minibatch to be propagated through the NN in the same forward execution, in other words, decreasing the need for "sub-minibatching" (Section 4.4.1). This sorting and chunking scheme generates minibatches that predominantly contain a single trace type. However, the minibatches used at each iteration are sampled randomly without replacement from different regions of the sorted training set, and therefore contain different trace types, resulting in a gradient unbiased in expectation during any given epoch.

In our initial profiling, the cost of I/O was more than 50% of total run time. With these data re-structuring and parallel I/O optimizations, we reduced the I/O to less than 5%, achieving 10x speedup at different scales.

4.4.4 Distributed improvements to PyTorch MPI CPU code. PyTorch has a torch.distributed backend,[15] which allows scalable distributed training with high performance on both CPU and GPU clusters. Etalumis uses the MPI backend as appropriate for the synchronous SGD setting that we implement (Algorithm 2) and the HPC machines we utilize (Section 5). We have made various improvements to this backend to enable the large-scale distributed training on CPU systems required for this project. The call `torch.distributed.all_reduce` is used to combine the gradient tensors for all distributed MPI ranks. In Etalumis, the set of non-null gradient tensors differs for each rank and is a small fraction of the total set of tensors. Therefore we first perform an allreduce to obtain a map of all the tensors that are present on all ranks; then we create a list of the tensors, filling in the ones that are not present on our rank with zero; finally, we reduce all of the gradient tensors in the list. PyTorch `all_reduce` does not take a list of tensors so normally a list comprehension is used, but this results in one call to `MPI_Allreduce` for each tensor. We modified PyTorch `all_reduce` to accept a list of tensors. Then, in the PyTorch C++ code for allreduce, we concatenate small tensors into a buffer, call `MPI_Allreduce` on the buffer, and copy the results back to the original tensor. This eliminates almost all the allreduce latency and makes the communication bandwidth-bound.

We found that changing Etalumis to reduce only the non-null gradients gives a 4x improvement in allreduce time. Tensor concatenation improves overall performance by an additional 4% on one node which increases as nodes are added. With these improvements, the load balance effects discussed in Sections 6.2 and 7.2 are dominant and so are our primary focus of further distributed optimizations. Other future work could include performing the above steps for each backward layer with an asynchronous allreduce to overlap the communications for the previous layer with the computation for the current layer.

5 SYSTEMS AND SOFTWARE

5.1 Cori

We use the "data" partition of the Cori system at the National Energy Research Scientific Computing Center (NERSC) at Lawrence Berkeley National Laboratory. Cori is a Cray XC40 system, and the data partition features 2,388 nodes. Each node has two sockets and each socket is populated with a 16-core 2.3 GHz Intel® Xeon® E5-2698 v3 CPU (referred to as HSW from now on), with peak single-precision (SP) performance of 1.2 Tflop/s and 128 GB of DDR4-2133 DRAM. Nodes are connected via the Cray Aries low-latency, high-bandwidth interconnect utilizing the dragonfly topology. In addition, the Cori system contains 288 Cray DataWarp nodes (also known as the "Burst Buffer") which house the input datasets for the Cori experiments presented here. Each DataWarp node contains 2 × 3.2 TB SSDs, giving a system total of 1.8 PB of SSD storage, with up to 1.7 TB/sec read/write performance and over 28M IOP/s. Cori also has a Sonnexion 2000 Lustre filesystem, which consists of 248 Object Storage Targets (OSTs) and 10,168 disks, giving nearly 30 PB of storage and a maximum of 700 GB/sec IO performance. This filesystem is used for output files (networks and logs) for Cori experiments and both input and output for the Edison experiments.

[14] Intel-optimized PyTorch: https://github.com/intel/pytorch

[15] https://pytorch.org/docs/stable/distributed.html

Table 1: Intel®Xeon® CPU models and codes

Model	Code
E5-2695 v2 @ 2.40GHz (12 cores/socket)	IVB
E5-2698 v3 @ 2.30GHz (16 cores/socket)	HSW
E5-2697A v4 @ 2.60GHz (16 cores/socket)	BDW
Platinum 8170 @ 2.10GHz (26 cores/socket)	SKL
Gold 6252 @ 2.10GHz (24 cores/socket)	CSL

5.2 Edison

We also make use of the Edison system at NERSC. Edison is a Cray XC30 system with 5,586 nodes. Each node has two sockets, each socket is populated with a 12-core 2.4 GHz Intel® Xeon® E5-2695 v2 CPU (referred to as IVB from now on), with peak performance of 460.8 SP Gflop/s, and 64 GB DDR3-1866 memory. Edison mounts the Cori Lustre filesystem described above.

5.3 Diamond cluster

In order to evaluate and improve the performance on newer Intel® processors we make use of the Diamond cluster, a small heterogeneous cluster maintained by Intel Corporation. The interconnect uses Intel® Omni-Path Architecture switches and host adapters. The nodes used for the results in this paper are all two socket nodes. Table 1 presents the CPU models used and the three letter abbreviations used in this paper.

5.4 Particle physics simulation software

In our experiments we use Sherpa version 2.2.3, coupled to a fast 3D detector simulator that we configure to use 20x35x35 voxels. Sherpa is implemented in C++, and therefore we use the C++ front end for PPX. We couple to Sherpa by a system-wide rerouting of the calls to the random number generator, which is made easy by the existence of a third-party random number generator interface (External_RNG) already present in Sherpa.

For this paper, in order to facilitate reproducible experiments, we run in the offline training mode and produce a sample of 15M traces that occupy 1.7 TB on disk. Generation of this 15M dataset was completed in 3 hours on 32 IVB nodes of Edison. The traces are stored using Python shelve[16] serialization, allowing random access to all entries contained in the collection of files with 100k traces in each file. These serialized files are accessed via the Python dbm module using the gdbm backend.

6 EXPERIMENTS AND RESULTS

6.1 Single node performance

We ran single node tests with one rank per socket for one and two ranks on the IVB nodes on Edison, the HSW partition of Cori and the BDW, SKL, and CSL nodes of the Diamond cluster. Table 2 shows the throughput and single socket flop rate and percentage of peak theoretical flop rate. We find that the optimizations described in Section 4.4.2 provide an improvement of 7x on the overall single socket run throughput (measured on HSW) relative to a default PyTorch version v1.0.0 installed via the official conda channel. We

[16]https://docs.python.org/3/library/shelve.html

Table 2: Single node training throughput in traces/sec and flop rate (Gflop/s). 1-socket throughput and flop rate are for a single process while 2-socket is for 2 MPI processes on a single node.

Platform	1-socket traces/s	2-socket traces/s	1-socket Gflop/s (% peak)
IVB (Edison)	13.9	25.6	196 (43%)
HSW (Cori)	32.1	56.5	453 (38%)
BDW (Diamond)	30.5	57.8	430 (32%)
SKL (Diamond)	49.9	82.7	704 (20%)
CSL (Diamond)	51.1	93.1	720 (22%)

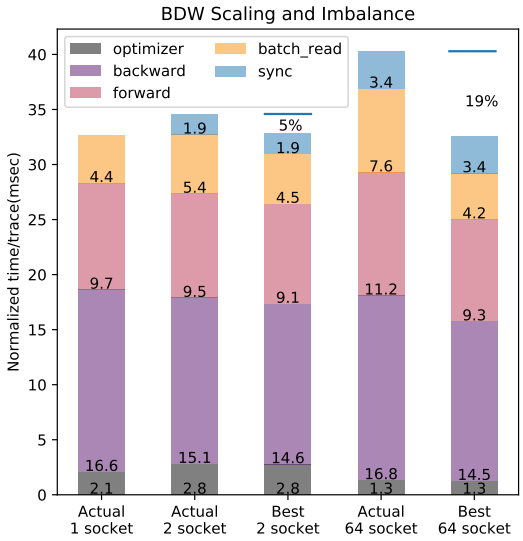

Figure 4: Actual and estimated best times for 1, 2, and 64 sockets. Horizontal bars at the top are to aid comparison between columns.

achieve **430 SP Gflop/s** on a single socket of the BDW system, measured using the available hardware counters for 256-bit packed SIMD single precision operations. This includes IO and is averaged over an entire 300k trace run. This can be compared to a theoretical peak flop rate for that BDW socket of 1,331 SP Gflop/s. Flop rates for other platforms are scaled from this measurement and given in Table 2. For further profiling we instrument the code with timers for each phase of the training (in order): minibatch read, forward, backward, and optimize. Figure 4 shows a breakdown of the time spent on a single socket after the optimizations described in Sections 4.4.1 and 4.4.2.[17]

6.2 Multi-node performance

In addition to the single socket operations, we time the two synchronization (allreduce) phases (gradient and loss). This information is recorded for each rank and each minibatch. Postprocessing finds the rank with the maximum work time (sum of the four phases

[17]See disclaimers section after conclusions.

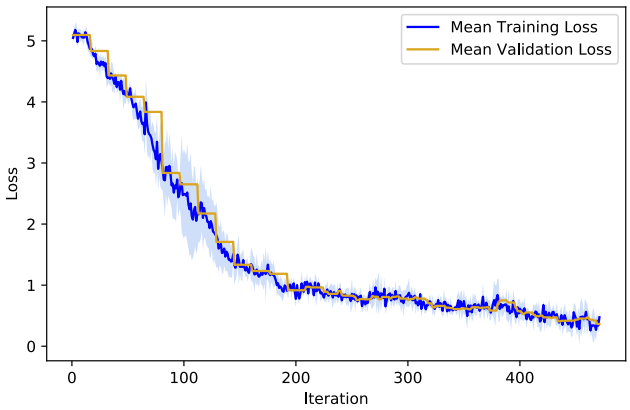

Figure 5: Mean loss and standard deviation (shaded) for five experiments with 128k minibatch size.

mentioned in Section 6.1) and adds the times together. This gives the actual execution time. Further, we compute the *average* time across ranks for each work phase for each minibatch and add those together, giving the best time assuming no load imbalance. The results are shown in Figure 4. Comparing single socket results with 2 and 64 socket results shows the increased impact of load imbalance as nodes are added. This demonstrates a particular challenge of this project where the load on each node depends on the random minibatch of traces sampled for training on that node. The challenge and possible mitigation approaches we explored are discussed further in Section 7.2.

6.3 Large scale training on Cori and Edison

In order to choose training hyperparameters, we explored global minibatch sizes of {64, 2k, 32k, 128k}, and learning rates in the range $\left[10^{-7}, 10^{-1}\right]$ with grid search and compared the loss values after one epoch to find the optimal learning rate for different global minibatch sizes separately. For global minibatch sizes of 2k, 32k, and 128k, we trained with both Adam and Adam-LARC optimizers and compared loss value changes as a function of iterations. For training at 1,024 nodes we choose to use 32k and 128k global minibatch sizes. For the 128k minibatch size, best convergence was found using the Adam-LARC optimizer with a polynomial decay (order=2) learning rate schedule [76] that decays from an initial global learning rate of 5.70×10^{-4} to final 2×10^{-5} after completing 12 epochs for the dataset with 15M traces. In Figure 5, we show the mean and standard-deviation for five training runs with this 128k minibatch size and optimizer, demonstrating stable convergence.

Figure 6 shows weak scaling results obtained for distributed training to over a thousand nodes on both the Cori and Edison systems. We use a fixed local minibatch size of 64 per rank with 2 ranks per node, and plot the mean and standard deviation throughput for each iteration in terms of traces/s (labeled "average" in the plot). We also show the fastest iteration (labeled "peak"). The average scaling efficiency at 1,024 nodes is 0.79 on Edison and 0.5 on Cori. The throughput at 1,024 nodes on Cori and Edison is 28,000 and 22,000 traces/s on average, with the peak as 42,000 and 28,000 traces/s respectively. One can also see that there is some variation in this performance due to the different compute times taken to process execution traces of different length and the related load imbalance as discussed in Sections 6.2 and 7.2. We determine the maximum sustained performance over a 10-iteration sliding window to be **450 Tflop/s** on Cori and 325 Tflop/s on Edison.[18]

We have performed distributed training with global minibatch sizes of 32k and 128k at 1,024-node scale for extended periods to achieve convergence on both Cori and Edison systems. This is illustrated in Figure 7 where we show the loss for training and validation datasets as a function of iteration for an example run on Edison.

6.4 Inference and science results

Using our framework and the NNs trained using distributed resources at NERSC as described previously, we perform inference on test τ observation data that has not been used for training. As the approach of applying probabilistic programming in the setting of large-scale existing simulators is completely novel, there is no direct baseline in literature that provides the full posterior in each of these variables. Therefore we use our own MCMC (RMH)-based posterior as a baseline for the validation of the IC approach. We establish the convergence of the RMH posterior by running two independent MCMC chains with different initializations and computing the the Gelman–Rubin convergence metric [24] to confirm that they converge onto the same posterior distribution (Section 4.2).

Figure 8 shows a comparison of inference results from the RMH and IC approaches. We show selected latent variables (addresses) that are a small subset of the more than 24k addresses that were encountered in the prior space of the Sherpa experimental setup, but are of physics interest in that they correspond to properties of the τ particle. It can be seen that there is close agreement between the RMH and IC posterior distributions validating that our network has been adequately trained. We have made various improvements to the RMH inference processing rate but this form of inference is compute intensive and takes **115 hours** on a Edison IVB node to produce the 7.68M trace result shown. The corresponding 6M trace IC result completed in **30 mins** (achieving a 230× speedup for a comparable posterior result) on 64 HSW nodes, enabled by the parallelism of IC inference.

In addition to parallelization, a significant advantage of the IC approach is that it is amortized. This means that once the proposal NN is trained for any given model, it can be readily applied to large volumes of new collision data. Moreover IC inference runs with high effective sample sizes in comparison to RMH: each sample from the IC NN is an independent sample from the proposal distribution, which approaches the true posterior distribution with more training, whereas our autocorrelation measurements in the RMH posterior indicate that a very large number of iterations are needed to get statistically independent traces (on the order of $\sim 10^5$ for the type of decay event we use as the observation). These features of IC inference combine to provide a tractable approach for fast Bayesian inference in complex models implemented by large-scale simulators.

[18] See disclaimers section after conclusions.

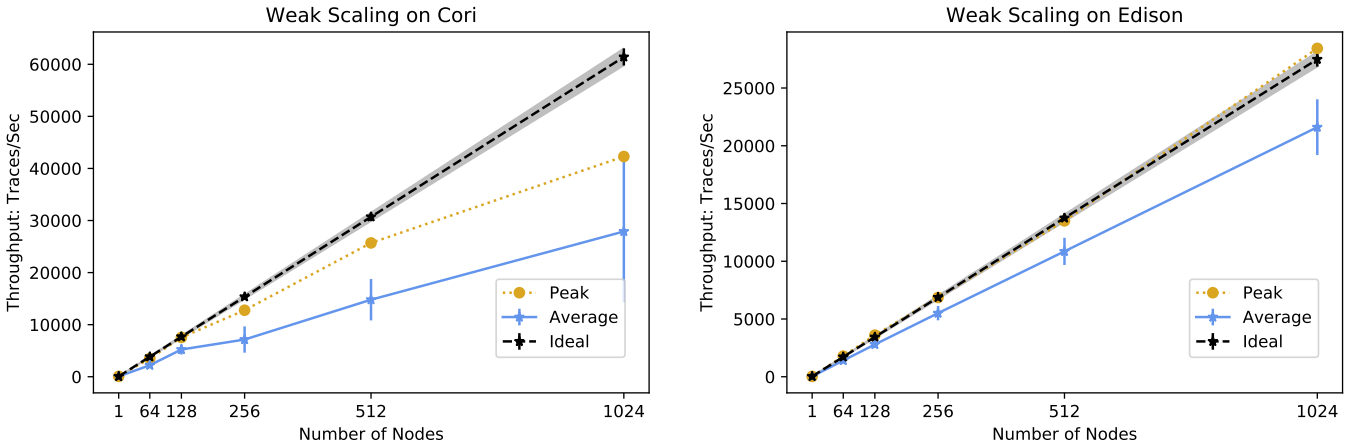

Figure 6: Weak scaling on Cori and Edison systems, showing throughput for different node counts with a fixed local minibatch size of 64 traces per MPI rank with 2 ranks per node. Average (mean) over all iterations and the peak single iteration are shown. Ideal scaling is derived from the mean single-node rate with a shaded uncertainty from the standard deviation in that rate.

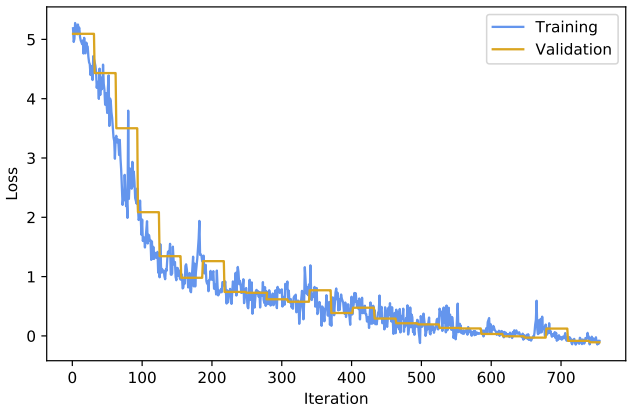

Figure 7: Training and validation loss for a 128k minibatch size experiment with the configuration described in the text run on 1,024 nodes of the Edison system.

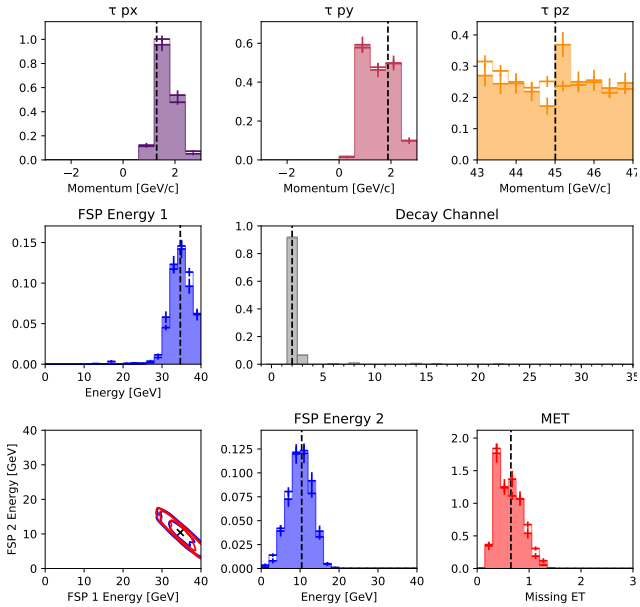

Figure 8: A comparison of posterior distributions obtained by RMH (filled histograms) and IC (outline histograms) and the ground truth values (dashed vertical lines) for a test τ decay observation. We show an illustrative subset of the latent variables, including x, y and z components of the τ-lepton momentum (top row), the energies of the two highest energy particles produced by the τ decay (middle left and bottom center), a contour plot showing correlation between these (bottom left), the τ decay channel ($\tau \rightarrow \pi \nu_\tau$ as mode) (middle right), and the missing transverse energy (bottom right).

7 DISCUSSION

The dynamic NN architecture employed for this project has presented a number of unique challenges for distributed training, which we covered in Section 4.4. Our innovations proved successful in enabling the training of this architecture at scale, and in this section we capture some of the lessons learned and unresolved issues encountered in our experiments.

7.1 Time to solution: trade-off between throughput and convergence

7.1.1 Increasing effective local minibatch size. As mentioned in Section 4.4.1, the distributed SGD scheme given in Algorithms 1 and 2 uses random traces sampled from the simulator, and can suffer from slow training throughput if computation cannot be

efficiently parallelized across the full minibatch due to the presence of different trace types. Therefore, we explored the following methods to improve effective minibatch size: sorting the traces before batching, online batching of the same trace types together, and multi-bucketing. Each of these methods can improve throughput, but risk introducing bias into SGD training or increasing the number of iterations to converge, so we compare the wall clock time for convergence to determine the relative trade-off. The impact of multi-bucketing is described and discussed in section 7.2 below. Sorting and batching traces from the same trace type together offers considerable throughput increase with a relatively small impact on convergence, when combined with shuffling of minibatches for randomness, so we used these throughput optimizations for the results in this paper.

7.1.2 Choice of optimizers, learning rate scaling and scheduling for convergence. There is considerable recent literature on techniques for distributed and large-minibatch training, including scaling learning rates with number of nodes [31], alternative optimizers for large-minibatch-size training, and learning rate decay [27, 41, 76]. The network employed in this project presents a very different use-case to those considered in literature, prompting us to document our experiences here. For learning rate scaling with node count, we found sub-sqrt learning rate scaling works better than linear for an Adam-based optimizer [41]. We also compared Adam with the newer Adam-LARC optimizer [27, 76] for convergence performance and found the Adam-LARC optimizer to perform better for the very large global minibatch size 128k in our case. For smaller global minibatch sizes of 32K or lower, both plain Adam and Adam-LARC performed equally well. Finally, for the learning rate decay scheduler, we explored the following options: no decay, multi-step decay (per epoch), and polynomial decay of order 1 or 2 (calculated per iteration) [76]. We found that learning-rate decay can improve training performance and polynomial decay of order 2 provided the most effective schedule.

7.2 Load balancing

As indicated in Section 6.2, our work involves distinct scaling challenges due to the variation in compute time depending on execution trace length, address-dependent proposal and embedding layers, and representation of trace types inside each minibatch. These factors contribute to load imbalance. The trace length variation bears similarity to varying sequence lengths in NMT; however, unlike that case it is not possible to truncate the execution traces, and padding would introduce a cost to overall number of operations.

To resolve this load imbalance issue, we have explored a number of options, building on those from NMT, including a *multi-bucketing* scheme and a novel *dynamic batching* approach.

In *multi-bucketing* [10, 20, 40], traces are grouped into several buckets based on lengths, and every global minibatch is solely taken from a randomly picked bucket for every iteration. Multi-bucketing not only helps to balance the load among ranks for the same iteration, but also increases the effective minibatch size as traces from the same trace type have a higher chance to be in the same minibatch than in the non-bucketing case, achieving higher throughput. For a local minibatch-size of 16 with 10 buckets we measured throughput increases in the range of 30–60% at 128–256 node scale on Cori. However, our current multi-bucketing implementation does multiple updates in the same bucket continuously. When this implementation is used together with batching the traces from the same trace type together (as discussed above in Section 7.1.1), it negatively impacts convergence behavior. We believe this to be due to the fact that training on each specific bucket for multiple updates introduces over-fitting onto that specific subset of networks so moving to a new bucket for multiple updates causes information of the progress made with previous buckets to be lost. As such we did not employ this configuration for the results reported in this paper.

With *dynamic batching*, we replaced the requirement of fixed minibatch size per rank with a desired number of "tokens" per rank (where a token is a unit of random number draws in each trace), so that we can, for instance, allocate many short traces (with smaller number of tokens each) for one rank but only a few long traces for another rank, in order to balance the load for the LSTM network due to length variation. While an equal-token approach has been used in NMT, this did not offer throughput gains for our model, which has an additional 3DCNN component in which the compute time depends on the number of traces within the local minibatch, so if dynamic batching only considers total tokens per rank for the LSTM it can negatively impact the 3DCNN load.

Through these experiments we found that our current optimal throughput and convergence performance came from not employing these load-balancing schemes although we intend to explore modifications to these schemes as ongoing work.

8 SCIENCE IMPLICATIONS AND OUTLOOK

We have provided a common interface to connect PPLs with simulators written in arbitrary code in a broad range of programming languages. This opens up possibilities for future work in all applied fields where simulators are used to model real-world systems, including epidemiology modeling such as disease transmission and prevention models [66], autonomous vehicle and reinforcement learning environments [21], cosmology [7], and climate science [68]. In particular, the ability to control existing simulators at scale and to generate interpretable posteriors is relevant to scientific domains where interpretability in model inference is critical.

We have demonstrated both MCMC- and IC-based inference of detector data originating from τ-decays simulated with the Sherpa Monte Carlo generator at scale. This offers, for the first time, the potential of Bayesian inference on the full latent structure of the large numbers of collision events produced at accelerators such as the LHC, enabling deep interpretation of observed events. For instance, ambiguity in the decay of a particle can be related exactly to the physics processes in the simulator that would give rise to that ambiguity. In order to fully realize this potential, future work will expand this framework to more complex particle decays (such as the Higgs decay to τ leptons) and incorporate a more detailed detector simulation (e.g., Geant4 [6]). We will demonstrate this on a full LHC physics analysis, reproducing the efficiency of point-estimates, together with the full posterior and intpretability, so that this can be exploited for discovery of new fundamental physics.

The IC objective is designed so that the NN proposal $q(\mathbf{x}|\mathbf{y})$ approximates the posterior $p(\mathbf{x}|\mathbf{y})$ asymptotically closely with more training. This costly training phase needs to be done *only once* for

a given simulator-based model, giving us a NN that can provide samples from the model posterior in parallel for any new observed data. In this setting where have a fast, amortized $q(\mathbf{x}|\mathbf{y}) \approx p(\mathbf{x}|\mathbf{y})$, our ultimate goal is to add the machinery of Bayesian inference to the toolbox for critical tasks such as triggering [1] and event reconstruction by conditioning on potentially interesting events (e.g., $q(\text{ParticleType}|\cdot) \geq \epsilon$). Recent activity exploring the use of FPGAs for NN inference for particle physics [23] will help implementation of these approaches, and HPC systems will be crucial in the training and inference phases of such frameworks.

9 CONCLUSIONS

Inference in simulator-based models remains a challenging problem with potential impact across many disciplines [14, 60]. In this paper we present the first probabilistic programming implementation capable of controlling existing simulators and running at large-scale on HPC platforms. Through the PPX protocol, our framework successfully couples with large-scale scientific simulators leveraging thousands of lines of existing simulation code encoding domain-expert knowledge. To perform efficient inference we make use of the inference compilation technique, and we train a dynamic neural network involving LSTM and 3DCNN components, with a large global minibatch size of 128k. IC inference achieved a 230× speedup compared with the MCMC baseline. We optimize the popular PyTorch framework to achieve a significant single-socket speedup for our network and 20–43% of peak theoretical flop rate on a range of current CPUs.[19] We augment and develop PyTorch's MPI implementation to run it at the unprecedented scale of 1,024 nodes (32,768 and 24,576 cores) of the Cori and Edison supercomputers with a sustained flop rate of 0.45 Pflop/s. We demonstrate we can successfully train this network to convergence at these large scales, and use this to perform efficient inference on LHC collision events. The developments described here open the door for exploiting HPC resources and existing detailed scientific simulators to perform rapid Bayesian inference in very complex scientific settings.

DISCLAIMERS

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

For more complete information visit www.intel.com/benchmarks.

Performance results are based on testing as of March 22 and March 28, 2019 and may not reflect all publicly available security updates. See configuration disclosure for details. No product or component can be absolutely secure.

Configurations: Testing on Cori and Edison (see §6.1) was performed by NERSC. Testing on Diamond cluster was performed by Intel.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at www.intel.com.

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.

Intel, VTune, Xeon are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

[19] See disclaimers section after conclusions.

ACKNOWLEDGMENTS

The authors would like to acknowledge valuable discussions with Thorsten Kurth on scaling aspects, Quincey Koziol on I/O; Steve Farrell on NERSC PyTorch, Holger Schulz on Sherpa, and Xiaoming Cui, from Intel AIPG team, on NMT. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231. This work was partially supported by the NERSC Big Data Center; we acknowledge Intel for their funding support. KC, LH, and GL were supported by the National Science Foundation under the awards ACI-1450310. Additionally, KC was supported by the National Science Foundation award OAC-1836650. BGH is supported by the EPRSC Autonomous Intelligent Machines and Systems grant. AGB and PT are supported by EPSRC/MURI grant EP/N019474/1 and AGB is also supported by Lawrence Berkeley National Lab. FW is supported by DARPA D3M, under Cooperative Agreement FA8750-17-2-0093, Intel under its LBNL NERSC Big Data Center, and an NSERC Discovery grant.

REFERENCES

[1] Morad Aaboud et al. 2017. Performance of the ATLAS Trigger System in 2015. *European Physical Journal* C77, 5 (2017), 317. https://doi.org/10.1140/epjc/s10052-017-4852-3

[2] G. Aad et al. 2008. The ATLAS Experiment at the CERN Large Hadron Collider. *JINST* 3 (2008), S08003. https://doi.org/10.1088/1748-0221/3/08/S08003

[3] G. Aad et al. 2015. Search for the Standard Model Higgs boson produced in association with top quarks and decaying into bb in pp collisions at sqrt s=8 TeV with the ATLAS detector. *The European Physical Journal C* 75, 7 (29 Jul 2015), 349. https://doi.org/10.1140/epjc/s10052-015-3543-1

[4] G. Aad et al. 2016. Reconstruction of hadronic decay products of tau leptons with the ATLAS experiment. *The European Physical Journal C* 76, 5 (25 May 2016), 295. https://doi.org/10.1140/epjc/s10052-016-4110-0

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

[6] Sea Agostinelli, John Allison, K al Amako, J Apostolakis, H Araujo, P Arce, M Asai, D Axen, S Banerjee, G Barrand, et al. 2003. GEANT4—a simulation toolkit. *Nuclear instruments and methods in physics research section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 506, 3 (2003), 250–303.

[7] Joël Akeret, Alexandre Refregier, Adam Amara, Sebastian Seehars, and Caspar Hasner. 2015. Approximate Bayesian computation for forward modeling in cosmology. *Journal of Cosmology and Astroparticle Physics* 2015, 08 (2015), 043.

[8] M Sanjeev Arulampalam, Simon Maskell, Neil Gordon, and Tim Clapp. 2002. A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. *IEEE Transactions on signal processing* 50, 2 (2002), 174–188.

[9] Atılım Güneş Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research (JMLR)* 18, 153 (2018), 1–43. http://jmlr.org/papers/v18/17-468.html

[10] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM, 41–48.

[11] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. 2018. Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research* (2018).

[12] Christopher M Bishop. 1994. *Mixture density networks*. Technical Report NCRG/94/004. Neural Computing Research Group, Aston University.

[13] Christopher M Bishop. 2006. *Pattern Recognition and Machine Learning*. Springer.

[14] Johann Brehmer, Gilles Louppe, Juan Pavez, and Kyle Cranmer. 2018. Mining gold from implicit models to improve likelihood-free inference. *arXiv preprint arXiv:1805.12244* (2018).

[15] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016. Revisiting distributed synchronous SGD. *arXiv preprint arXiv:1604.00981* (2016).

[16] D. Das, S. Avancha, D. Mudigere, K. Vaidynathan, S. Sridharan, D. Kalamkar, B. Kaul, and P. Dubey. 2016. Distributed Deep Learning Using Synchronous Stochastic Gradient Descent. *ArXiv e-prints* (Feb. 2016). arXiv:cs.DC/1602.06709

[17] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12)*. Curran Associates Inc., USA, 1223–1231. http://dl.acm.org/citation.cfm?id=2999134.2999271

[18] Adji Bousso Dieng, Dustin Tran, Rajesh Ranganath, John Paisley, and David Blei. 2017. Variational Inference via χ Upper Bound Minimization. In *Advances in Neural Information Processing Systems*. 2732–2741.

[19] Joshua V Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matt Hoffman, and Rif A Saurous. 2017.

TensorFlow distributions. *arXiv preprint arXiv:1711.10604* (2017).
[20] Patrick Doetsch, Pavel Golik, and Hermann Ney. 2017. A comprehensive study of batch construction strategies for recurrent neural networks in mxnet. *arXiv preprint arXiv:1705.02414* (2017).
[21] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*. 1–16.
[22] Arnaud Doucet and Adam M Johansen. 2009. A tutorial on particle filtering and smoothing: Fifteen years later. (2009).
[23] Javier Duarte et al. 2018. Fast inference of deep neural networks in FPGAs for particle physics. *JINST* 13, 07 (2018), P07027. https://doi.org/10.1088/1748-0221/13/07/P07027 arXiv:physics.ins-det/1804.06913
[24] Andrew Gelman, Hal S Stern, John B Carlin, David B Dunson, Aki Vehtari, and Donald B Rubin. 2013. *Bayesian data analysis*. Chapman and Hall/CRC.
[25] Samuel Gershman and Noah Goodman. 2014. Amortized inference in probabilistic reasoning. In *Proceedings of the annual meeting of the cognitive science society*, Vol. 36.
[26] Zoubin Ghahramani. 2015. Probabilistic machine learning and artificial intelligence. *Nature* 521, 7553 (2015), 452.
[27] B. Ginsburg, I. Gitman, and O. Kuchaiev. 2018. Layer-Wise Adaptive Rate Control for Training of Deep Networks. *in preparation* (2018).
[28] Sheldon L Glashow. 1961. Partial-symmetries of weak interactions. *Nuclear Physics* 22, 4 (1961), 579–588.
[29] Tanju Gleisberg, Stefan Höche, F Krauss, M Schönherr, S Schumann, F Siegert, and J Winter. 2009. Event generation with SHERPA 1.1. *Journal of High Energy Physics* 2009, 02 (2009), 007.
[30] Prem Gopalan, Wei Hao, David M Blei, and John D Storey. 2016. Scaling probabilistic models of genetic variation to millions of humans. *Nature genetics* 48, 12 (2016), 1587.
[31] Priya Goyal, Piotr Dollar, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv preprint arXiv:1706.02677v1* (2017).
[32] David Griffiths. 2008. *Introduction to elementary particles*. John Wiley & Sons.
[33] Ralf Herbrich, Tom Minka, and Thore Graepel. 2007. TrueSkill™: a Bayesian skill rating system. In *Advances in neural information processing systems*. 569–576.
[34] Pieter Hintjens. 2013. *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc.
[35] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[36] Matthew D Hoffman, David M Blei, Chong Wang, and John Paisley. 2013. Stochastic variational inference. *The Journal of Machine Learning Research* 14, 1 (2013), 1303–1347.
[37] Matthew D Hoffman and Andrew Gelman. 2014. The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research* 15, 1 (2014), 1593–1623.
[38] F. N. Iandola, K. Ashraf, M. W. Moskewicz, and K. Keutzer. 2015. FireCaffe: near-linear acceleration of deep neural network training on compute clusters. *ArXiv e-prints* (Oct. 2015). arXiv:cs.CV/1511.00175
[39] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2016. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836* (2016).
[40] Viacheslav Khomenko, Oleg Shyshkov, Olga Radyvonenko, and Kostiantyn Bokhan. 2016. Accelerating recurrent neural network training using sequence bucketing and multi-gpu data parallelization. In *2016 IEEE First International Conference on Data Stream Mining & Processing (DSMP)*. IEEE, 100–103.
[41] D. P. Kingma and J. Ba. 2014. Adam: A Method for Stochastic Optimization. *ArXiv e-prints* (Dec. 2014). arXiv:cs.LG/1412.6980
[42] Diederik P Kingma and Max Welling. 2013. Auto-encoding variational Bayes. *arXiv preprint arXiv:1312.6114* (2013).
[43] Kunitaka Kondo. 1988. Dynamical likelihood method for reconstruction of events with missing momentum. I. Method and toy models. *Journal of the Physical Society of Japan* 57, 12 (1988), 4126–4140.
[44] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, Prabhat, and Michael Houston. 2018. Exascale Deep Learning for Climate Analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 51, 12 pages. http://dl.acm.org/citation.cfm?id=3291656.3291724
[45] Thorsten Kurth, Jian Zhang, Nadathur Satish, Evan Racah, Ioannis Mitliagkas, Md Mostofa Ali Patwary, Tareq Malas, Narayanan Sundaram, Wahid Bhimji, Mikhail Smorkalov, et al. 2017. Deep learning at 15PF: supervised and semi-supervised classification for scientific data, In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. *ArXiv e-prints*, 7. arXiv:1708.05256
[46] Tuan Anh Le. 2015. Inference for higher order probabilistic programs. *Masters thesis, University of Oxford* (2015).
[47] Tuan Anh Le, Atılım Güneş Baydin, and Frank Wood. 2017. Inference Compilation and Universal Probabilistic Programming. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS) (Proceedings of Machine Learning Research)*, Vol. 54. PMLR, Fort Lauderdale, FL, USA, 1338–1348.
[48] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
[49] Mario Lezcano Casado, Atılım Güneş Baydin, David Martinez Rubio, Tuan Anh Le, Frank Wood, Lukas Heinrich, Gilles Louppe, Kyle Cranmer, Wahid Bhimji, Karen Ng, and Prabhat. 2017. Improvements to Inference Compilation for Probabilistic Programming in Large-Scale Scientific Simulators. In *Neural Information Processing Systems (NIPS) 2017 workshop on Deep Learning for Physical Sciences (DLPS), Long Beach, CA, US, December 8, 2017.*
[50] Linux man-pages project. 2019. *Linux Programmer's Manual*. http://man7.org/linux/man-pages/index.html
[51] Amrita Mathuriya, Deborah Bard, Peter Mendygral, Lawrence Meadows, James Arnemann, Lei Shao, Siyu He, Tuomas Karna, Daina Moise, Simon J. Pennycook, Kristyn Maschoff, Jason Sewall, Nalini Kumar, Shirley Ho, Mike Ringenburg, Prabhat, and Victor Lee. 2018. CosmoFlow: Using Deep Learning to Learn the Universe at Scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 65, 11 pages. https://dl.acm.org/citation.cfm?id=3291743
[52] Amrita Mathuriya, Thorsten Kurth, Vivek Rane, Mustafa Mustafa, Lei Shao, Debbie Bard, Victor W Lee, et al. 2017. Scaling GRPC Tensorflow on 512 nodes of Cori Supercomputer. In *Neural Information Processing Systems (NIPS) 2017 workshop on Deep Learning At Supercomputer Scale, Long Beach, CA, US, December 8, 2017.*
[53] Sam McCandlish, Jared Kaplan, and et.al Amodei, Dario. 2018. An Empirical Model of Large-Batch Training. *arXiv preprint arXiv:1812.06162v1* (2018).
[54] Hiroaki Mikami, Hisahiro Suganuma, Pongsakorn U.-Chupala, Yoshiki Tanaka, and Yuichi Kageyama. 2018. ImageNet/ResNet-50 Training in 224 Seconds. *CoRR* abs/1811.05233 (2018). arXiv:1811.05233 http://arxiv.org/abs/1811.05233
[55] T. Minka, J.M. Winn, J.P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. 2018. /Infer.NET 0.3. Microsoft Research Cambridge. http://dotnet.github.io/infer.
[56] Radford M Neal. 1993. *Probabilistic inference using Markov chain Monte Carlo methods*. Technical Report CRG-TR-93-1. Dept. of Computer Science, University of Toronto.
[57] Radford M Neal et al. 2011. MCMC using Hamiltonian dynamics. *Handbook of markov chain monte carlo* 2, 11 (2011), 2.
[58] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. 2011. HOGWILD!: A Lock-free Approach to Parallelizing Stochastic Gradient Descent. In *Proceedings of the 24th International Conference on Neural Information Processing Systems (NIPS'11)*. Curran Associates Inc., USA, 693–701. http://dl.acm.org/citation.cfm?id=2986459.2986537
[59] X. Pan, J. Chen, R. Monga, S. Bengio, and R. Jozefowicz. 2017. Revisiting Distributed Synchronous SGD. *ArXiv e-prints* (Feb. 2017). arXiv:cs.DC/1702.05800
[60] George Papamakarios and Iain Murray. 2016. Fast ϵ-free Inference of Simulation Models with Bayesian Conditional Density Estimation. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). Curran Associates, Inc., 1028–1036.
[61] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS 2017 Autodiff Workshop: The Future of Gradient-based Machine Learning Software and Techniques, Long Beach, CA, US, December 9, 2017.*
[62] Michael E Peskin. 2018. *An introduction to quantum field theory*. CRC Press.
[63] A Salam. 1968. Proceedings of the Eighth Nobel Symposium on Elementary Particle Theory, Relativistic Groups, and Analyticity, Stockholm, Sweden, 1968. (1968).
[64] Christopher J. Shallue, Jaehoom Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. 2018. Measuring the Effects of Data Parallelism on Neural Network training. *arXiv preprint arXiv:1811.03600v2* (2018).
[65] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. 2017. Don't Decay the Learning Rate, Increase the Batch Size. *arXiv preprint arXiv:1711.00489* (2017).
[66] T. Smith, N. Maire, A. Ross, M. Penny, N. Chitnis, A. Schapira, A. Studer, B. Genton, C. Lengeler, F. Tediosi, and et al. 2008. Towards a comprehensive simulation model of malaria epidemiology and control. *Parasitology* 135, 13 (2008), 1507–1516. https://doi.org/10.1017/S0031182008000371
[67] Sam Staton, Frank Wood, Hongseok Yang, Chris Heunen, and Ohad Kammar. 2016. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1–10.
[68] John Sterman, Thomas Fiddaman, Travis Franck, Andrew Jones, Stephanie McCauley, Philip Rice, Elizabeth Sawin, and Lori Siegel. 2012. Climate interactive: the C-ROADS climate policy model. *System Dynamics Review* 28, 3 (2012), 295–305.

[69] Michael Teng and Frank Wood. 2018. Bayesian Distributed Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems*. 6380–6390.

[70] Dustin Tran, Alp Kucukelbir, Adji B Dieng, Maja Rudolph, Dawen Liang, and David M Blei. 2016. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787* (2016).

[71] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming. *arXiv e-prints*, Article arXiv:1809.10756 (Sep 2018). arXiv:stat.ML/1809.10756

[72] Martinus Veltman et al. 1972. Regularization and renormalization of gauge fields. *Nuclear Physics B* 44, 1 (1972), 189–213.

[73] S Weinberg. 1967. *Phys. Rev. Lett* 19 (1967), 1264.

[74] David Wingate, Andreas Stuhlmüller, and Noah Goodman. 2011. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 770–778.

[75] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, and et. al. Norouzi Mohammad. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. (10 2016).

[76] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888* (2017).

[77] S. Zheng, Q. Meng, T. Wang, W. Chen, N. Yu, Z.-M. Ma, and T.-Y. Liu. 2016. Asynchronous Stochastic Gradient Descent with Delay Compensation. *ArXiv e-prints* (Sept. 2016). arXiv:cs.LG/1609.08326