High Throughput Training of Deep Surrogates from Large Ensemble Runs

Lucas Meyer, Marc Schouler, Robert Alexander Caulk, Alejandro Ribés, Bruno Raffin

▶ **To cite this version:**

Lucas Meyer, Marc Schouler, Robert Alexander Caulk, Alejandro Ribés, Bruno Raffin. High Throughput Training of Deep Surrogates from Large Ensemble Runs. SC 2023 - The International Conference for High Performance Computing, Networking, Storage, and Analysis, Nov 2023, Denver, CO, United States. pp.1-14, 10.1145/3581784.3607083 . hal-04213978

HAL Id: hal-04213978
https://hal.science/hal-04213978

Submitted on 28 Sep 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

High Throughput Training of Deep Surrogates from Large Ensemble Runs

Lucas Meyer
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG
Grenoble, France
Industrial AI Laboratory SINCLAIR, EDF Lab Paris-Saclay
Palaiseau, France

Marc Schouler
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG
Grenoble, France

Robert Alexander Caulk
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG
Grenoble, France

Alejandro Ribes
Industrial AI Laboratory SINCLAIR, EDF Lab Paris-Saclay
Palaiseau, France

Bruno Raffin
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG
Grenoble, France

ABSTRACT

Recent years have seen a surge in deep learning approaches to accelerate numerical solvers, which provide faithful but computationally intensive simulations of the physical world. These deep surrogates are generally trained in a supervised manner from limited amounts of data slowly generated by the same solver they intend to accelerate. We propose an open-source framework that enables the online training of these models from a large ensemble run of simulations. It leverages multiple levels of parallelism to generate rich datasets. The framework avoids I/O bottlenecks and storage issues by directly streaming the generated data. A training reservoir mitigates the inherent bias of streaming while maximizing GPU throughput. Experiment on training a fully connected network as a surrogate for the heat equation shows the proposed approach enables training on 8TB of data in 2 hours with an accuracy improved by 47% and a batch throughput multiplied by 13 compared to a traditional offline procedure.

CCS CONCEPTS

• **Computing methodologies** → **Online learning settings**; *Distributed artificial intelligence*; **Massively parallel and high-performance simulations**; • **Applied computing** → *Physical sciences and engineering*.

KEYWORDS

Online Deep Learning, Numerical Simulations, Large Scale Ensemble Run, In Situ Analysis, Parallel Training

ACM Reference Format:
Lucas Meyer, Marc Schouler, Robert Alexander Caulk, Alejandro Ribes, and Bruno Raffin. 2023. High Throughput Training of Deep Surrogates from Large Ensemble Runs. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3581784.3607083

SC23, November 12–17, 2023, Denver, CO
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA*, https://doi.org/10.1145/3581784.3607083.

1 INTRODUCTION

The interest in integrating deep neural networks with traditional numerical simulations has risen in recent years [19, 31, 36, 75]. The goal is to accelerate numerical simulations crucial to many scientific and engineering applications [2, 55]. These simulations generally provide a faithful representation of complex physical phenomena at the cost of intensive computation. The numerical simulation, typically a partial differential equation solver, is a process f that takes as input X, a vector that encompasses the parameters of the equation, and produces u_X^t a discretized time series of the different time steps of the solution (Equation 1):

$$\begin{aligned} f &: \mathbb{R}^{d_{\text{in}}} \rightarrow \mathbb{R}^{d_{\text{out}}+1} \\ X &\mapsto \{u_X^t\}_{0 \leq t < \tau} \end{aligned} \tag{1}$$

One of the common machine learning approaches is to design and train a model f_θ, referred to as a *deep surrogate*, that approximates the process f. f_θ is not expected to have the same generalization capabilities as the original solver, but rather, for a given range of parameters, produce much faster simulations.

The deep surrogate can then be used to identify an optimal configuration (e.g. identifying designs in fluid interaction [6]). It can also serve to estimate a probability density function or associated statistics [43], perform bayesian inference tasks [21], and inverse problems [64]. Machine learning approaches have been reported to be several orders of magnitude faster for such tasks than traditional solvers [38, 40, 82]. Deep surrogates are also lighter than traditional solvers and the massive simulation data they produce. These architectures can be seen as efficient compression methods for *post hoc* data analysis and visualization [28, 51]. Besides, being developed with deep learning frameworks that support automatic differentiation, these surrogates directly provide different gradients, including the adjoint, valuable for many applications [63].

Generally, the training of these deep surrogates is supervised. It requires the generation of a dataset of simulations. The good generalization capabilities of the surrogate model depend on both the neural architecture and the training dataset [19, 41, 70]. To

create a dataset representative of all the richness the process f exhibits when X varies requires executing the solver many times (*i.e.* ensemble runs). The training dataset can quickly grow in size to prohibitive extents, making it difficult to store on disks and incurring substantial I/O operations, which hinders the training speed. Additionally, training cannot adapt dynamically the data generation process, the latter being performed *a priori*.

This paper focuses on the combined data generation and training of deep surrogate models. Classical training procedures rely on a fixed dataset that is stored on disk and read to extract batches. However, because deep surrogates are trained with synthetic data produced by simulation code, the training can be performed online, simultaneously with the data generation. The potential benefits are:

- **Storage avoiding**. The data are never stored on disk, saving storage space. On supercomputers, storage is often limited, while the increasing size of training datasets exacerbates the demand for storage space and i-nodes.
- **I/O bypass**. Directly transferring the data from the simulation to the neural network bypasses storage and circumvents the I/O bottleneck. I/O slows down both the data generation when writing data to disk, as well as training when reading back data from disk. This is a well-known issue in HPC that led to the in situ data processing paradigm [12].
- **Training diversity**. Because the data are produced by a simulation code, the deep surrogate can potentially be trained from an unlimited dataset, exposing the surrogate to more diverse data than with a fixed dataset repeated over several training epochs. At an equivalent batch count, this higher data diversity can enable online training to converge faster and reach better generalization capabilities compared to offline epoch-based training.

The contribution of the present paper is an online and large scale training framework for deep surrogate models called Melissa [71]. It combines:

(1) A multi-level parallelism (parallel solver execution, distributed data parallel training, ensemble run execution), in transit data processing bypassing storage, fault-tolerance for resilience, heterogeneous architecture support, and elasticity for adaptive executions, all features required to ensure efficient training at scale.
(2) A training reservoir. A buffer that mitigates the inherent bias in the streamed data caused by the solver internal logic and the availability of computational resources. The reservoir optimizes the throughput of data presented to the GPUs for training while maximizing their diversity.

We show through experiments that the resulting framework is capable of leveraging multiple levels of parallelism to efficiently train deep surrogates in an online context. It generates and trains a neural network on 8TB of simulation data in less than 2 hours using 5,000 cores and 4 GPUs. Such a dataset, which could hardly be handled on a cluster in a traditional offline training setting, would require more than 24 hours to be processed by the same number of GPUs. Compared to offline training with multiple epochs on a subset of 100GB, online training increases by 47% the generalization capability of the trained surrogate at an equivalent number of batches.

Section 2 discusses the related work. Section 3 presents the architecture and the developed strategies. Section 4 follows with the experiments, while Section 5 concludes this paper.

2 RELATED WORK

2.1 Deep surrogates for numerical simulations

Recent research on deep surrogate modeling focuses on identifying and training suitable neural architectures for accelerating numerical solutions, while aiming to maintain consistency with physical laws. Most of the deep surrogates found in the literature are trained in a supervised manner from simulation data. The seminal work of Raissi et al. on Physics-Informed Neural Networks (PINNs) could stand as an exception [64], as they can be trained unsupervised. But PINNs also benefit from training with simulation data [41, 49].

When the simulation domain is a regular mesh, time steps u_X^t can be seen as images and convolutional networks can be employed successfully [37, 38, 67, 81, 91]. The case of irregular meshes can be addressed with specific architectures like Graph Neural Networks (GNNs)[18, 61] or approached with Fourier neural operators [46]. Some have integrated the time dimension using recurrent architectures [77] or attention mechanisms [47].

Not only does the space discretization influence the design of the architectures, but the handling of the time dimension also distinguishes *autoregressive* from *direct* models. Autoregressive models mimic the iterative process of traditional solvers, where the current state is used as input to predict the next one ($f_\theta(u_X^{t-1}) \approx u_X^t$). One challenge for autoregressive models is the error accumulation along a trajectory, leading to various mitigation strategies [18, 61, 76]. Despite this challenge, the autoregressive approach produces versatile deep surrogates that are, in theory, not limited to the time range they have been trained on. Direct models produce the state corresponding to the time step provided as input ($f_\theta(X, t) \approx u_X^t$). PINNs are an example of direct models that are trained by minimizing the residual error of the PDE at random collocation points [64, 74, 80].

This paper presents experiments for supervised training of direct deep surrogates. Nonetheless, the presented framework supports the training of any deep surrogate provided it relies on simulation data. For instance, it has been employed to train autoregressive models at a small scale with a less advanced buffering algorithm [53].

2.2 Online deep learning

In the machine learning context, *online training* often refers to long training for which the probability distribution of the presented data shifts over time [33, 59, 69]. How this distribution shift occurs depends on the application. Typically, *lifelong training* considers training for very long periods of time as it occurs in recommendation systems. *Streaming learning* characterizes training for which samples arrive continuously and are processed individually. In this paper, the online characterization denotes the simultaneity of the generation of synthetic data, which can be controlled, with the training of the model.

The probability distribution shift can lead to *catastrophic forgetting* where the deep learning architecture trained on recent data

sees a deterioration of its capabilities on old data that is no longer or much less present in the training set.

Deep Reinforcement Learning (Deep RL) is another domain where online training is common. It involves actors that interact with a simulation according to a deep learning architecture that implements the action policy. Several actor instances are executed to produce trajectories that directly feed a learner trained to improve the current policy. *Replay buffers* are commonly used as intermediate temporary storage between actors and the learner to mitigate bias and catastrophic forgetting [26, 34, 65, 89]. Actor concurrency also contributes to better data diversity [84]. Solutions developed to orchestrate the online training of the learner on data generated by various actors are specific to Deep RL. They involve specific management of off-policy training, *i.e.* training with trajectories generated under outdated policies. These considerations are irrelevant to the training of deep surrogates because physical laws are constant and never outdated as Deep RL policy can become. This paper reuses the idea of an intermediate buffer but tailored to the context of deep surrogates training. Additionally, it is important to note that even though Deep RL training can require massive distributed resources [14, 57, 73], the simulation code is not, to our knowledge, computationally intensive according to HPC standards. It runs on a single node, sometimes using a GPU. Thus frameworks for distributed Deep RL, like RLlib [48], do not face the additional complexity of working with simulation codes parallelized for distributed memory.

2.3 Simulation ensemble management

Efficient management of large ensembles on supercomputers has been a subject of investigation for a long time in applications like sensitivity analysis or data assimilation. The most direct approach relies on files to store intermediate results [7, 11, 25, 50, 58]. Thus, each *member* of the ensemble, *i.e.* instance of the simulation to run, can be executed independently. Data processing is triggered once all members have been executed. Fault tolerance is easily enabled, but relying on files can impact performance. Using on-node storage sometimes available on supercomputers can contribute to reducing the I/O bottleneck [86].

The second standard approach consists in assembling all components of the workflow in a single large MPI application [7, 58]. This is particularly used for data assimilation, which works with cycles of propagation and updates. The members propagate the simulation states for some time steps, then these states are gathered and corrected using observations. Once corrected they are redistributed to the members for them to proceed with the next batch of times steps. If intermediate files are avoided, fault tolerance and load balancing become challenging. These important features, especially when targeting the very large scale, are seldom supported with such an approach.

Intermediate models have been more recently explored. Members process data online relying on dynamic client/server $N \times M$ data communications [17, 27, 79]. These intermediate models keep the best of both worlds: the efficiency of a file-avoiding solution while retaining the necessary flexibility to support fault tolerance, load balancing and some elasticity. The framework presented in the paper adopts this approach, extending the Melissa framework initially developed for sensibility analysis [79].

2.4 Task and workflow

Ensemble runs are a specific type of workflow, often developed with distributed task-based environments or workflow managers. Examples of these environments and managers include Ray, Dask, Parsl, Pycompss, RADICAL-Cybertool, qgc-pilot [8, 10, 15, 56, 66, 78]. Few are actually capable of enabling $N \times M$ dynamic connections between legacy MPI parallel tasks while ensuring fault tolerance.

The possibilities of exascale computing to open new scientific opportunities through large ensembles has been stressed early for molecular dynamics [62]. Deep surrogates are relevant in pure numerical schemes, but also in workflows combined with other scientific instruments [1, 5]. Some are reporting gains of several orders of magnitude[44, 82] when assessing globally the cost of deep surrogate training and the gains of using the surrogate versus the original simulation. The basic workflow sequences two steps: 1) surrogate training, 2) surrogate inference for addressing the target problem, potentially combined with some simulation runs when higher precision is needed. But some are pushing the logic one step further fusing these two steps into a single *adaptive ensemble run* where a steering logic, relying on shallow or deep learning, tries to improve the global workflow efficiency [9, 83, 88]. In this paper we focus on the deep surrogate training process (step 1), but our approach has all the necessary flexibility to be used in the fused workflow.

To conclude this section, we mention emerging approaches combining ensemble runs and deep learning, like simulation-based inference [13, 21] or simulation intelligence [42], that further stress the growing potential of deep surrogates and online training.

3 FRAMEWORK

The presented framework aims to optimize the throughput of data generation, transmission, batch creation, and distributed training for deep learning at a large scale (thousands of CPUs and multiple GPUs). At its core, the design leverages the stochastic nature of gradient descent, where the data presented to a Neural Network (NN) for training are sampled according to a given density distribution (usually uniform) and not strictly ordered. The framework exploits these loose synchronization and data ordering requirements to improve large-scale resource usage. In the following, Section 3.1 presents the framework's components and their assemblies, then Section 3.2 provides the details of data management and buffering algorithms.

3.1 Architecture

The framework architecture relies on a client/server model extended to the parallel case where both the client and server are programs with potentially different levels of parallelism (Figure 1). No intermediate file is required as all data exchanges take place through direct memory-to-memory communications between the clients and the server.

This client/server architecture improves the application modularity. Because the connection between a client and the server is dynamic, a client can be stopped (voluntarily or not) and started

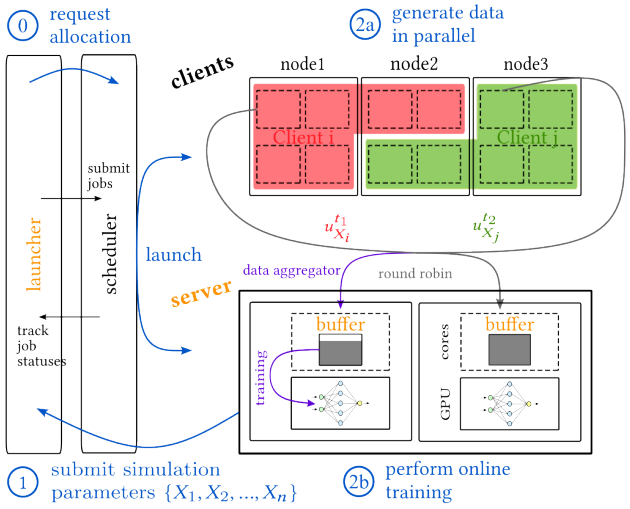

Figure 1: Framework architecture. Core components are highlighted in orange. The different steps of the workflow are represented in blue. The data generation (2a) and the training (2b) occur simultaneously. Here, 2 clients, *i.e.* 2 simulation instances of respective parameters X_i and X_j, run on 6 cores each, spanning over a total of 3 nodes. Training is performed by the server with distributed data parallelism on 2 GPUs. As soon as time steps (e.g. $u_{X_i}^{t_1}$ and $u_{X_j}^{t_2}$) are computed by the clients, they are streamed to the server. On each process of the server a data aggregator thread polls for new data to store in the buffer. Concurrently, the training thread extracts batches from the buffer and proceeds with training.

anytime. A client failure does not lead the server to failure, providing a sound base to support an efficient fault tolerance protocol. The number of running clients can evolve with time according to the resources available on the supercomputer, making the application elastic.

Each **client** runs an instance of the simulation code f with different input parameters X. The simulation is often an MPI+X parallel code running on several cores and nodes. As soon as a client produces a new time step u_X^t, this one is sent to the server via the API. Because clients run as independent executables, each one can use different types or amounts of resources.

The **server** is in charge of training. It is an MPI code relying on distributed data parallelism for parallel training. All MPI processes run an identical copy of the NN, but each one trains it with different data. After each batch backpropagation, the locally computed vector of weight updates is all-reduced between all processes and applied to each local NN copy to keep them identical [45]. This is today the standard parallelization approach for training, capable of using thousands of GPUs if learning rate and batch sizes are managed properly [30, 54]. The framework has not yet been experimented with model parallelism [35, 85]. Model parallelism is often combined with data parallelism when the NN cannot fit into the memory of one GPU. This would require splitting the NN and the batches to enable pipeline parallelism for instance as well as making parallel all-reduces per group of GPUs managing the same sub-part of the NN [72]. In the context of our framework, changes are expected to be mainly limited to the code of the training thread. Today, DL frameworks like Pytorch or TensorFlow, on which we rely, provide extensions to ease the deployment of multi-GPU model parallelism.

Each server process runs two threads. The **data aggregator thread** manages connections to clients, receives data and stores these data into the training buffer. The second thread, the **training thread**, reads data from the training buffer to build a batch, feeds the GPU with it and performs the forward and backward passes through the NN. An all-reduce operation amongst the different training threads aggregates the gradients to update the network weights. Finally, each thread copy of the network is updated locally before repeating the process with a new batch.

The data aggregator thread controls the **experimental design**. Methods currently supported to draw the parameters X for each client include the traditional Monte Carlo method, Latin hypercube and Halton sequence. Because the server drives the training progress, the experimental design could be made adaptive to support *active learning* strategies.

There is one **training buffer** per server process. This is a thread-safe data container of fixed capacity shared between the aggregator and the learner thread. This buffer, discussed in detail in Section 3.2.3, is a critical component to balance the quality and speed of training.

The **launcher** orchestrates and monitors the workflow. The launcher interacts with the supercomputer batch scheduler to start clients or server jobs, monitor their progress, kill some of them or restart them in case of failure. The launcher first starts the server job. Next, the server forwards to the launcher requests for executing client instances. The launcher takes care of building the associated jobs and submits them to the batch scheduler. The launcher's default behavior is to request one resource allocation for the server and one per client. But this approach shows limitations in two cases. First, when the client run is very short or just requires a few cores, the scheduling overheads dominate leading to idleness on the server side. Next, as all jobs are independent their start time depends on resource availability and may also lead to server idleness when the number of running clients is too low. To mitigate these issues, the framework provides a schedule-in-schedule approach where a larger resource allocation is requested (or several large ones) and jobs are scheduled into this allocation. Currently, the framework supports the Slurm [87] and OAR [20] schedulers. Integration of workflow schedulers like RadicalPilot [52] or Flux [3], to directly and efficiently take care of this two-level scheduling scheme, is planned.

The workflow is heterogeneous, usually running clients on CPU nodes and the server on GPU nodes, each type of node being managed independently through two different scheduling queues. As highlighted by experiments (Section 4), much fewer GPUs are usually needed compared to the number of CPUs. As the server starts first, it is natural to request first a reservation of GPU nodes and next CPU nodes for clients. However, the CPU partition was significantly more loaded than the GPU one, leading to a server staying idle for long periods while waiting for CPU resources. We thus had to reverse the reservation scheme, requesting first CPU resources, and, once available, GPU resources for the server. This proved to

be the most economical approach to preserve our compute hour budget. Notice that schedulers like Slurm can support directly such heterogeneous jobs, but we were asked not to use this feature as it apparently could affect the resource allocation efficiency.

The framework is **fault-tolerant**. The server watches for unresponsive clients and asks the launcher to properly kill and restart faulty ones. The server maintains a log of received messages per client, so in case of client restart, already received messages are discarded. If the client simulation code supports checkpointing, it can be enabled so the client will restart from the last checkpoint only. The server is regularly checkpointed. If a server failure is detected by the launcher, it first kills all running clients and next restarts a new server instance from the last checkpoint. This server will request the launcher to restart the necessary client instances. When the launcher fails, the currently running clients continue until completion, after which the server checkpoints and stops. It has then to be restarted manually.

The framework, Melissa, is open source[1]. All the stochastic components (*i.e.* the network weight initialization, the simulation parameter sampler, and the training buffer) are seeded for reproducibility purposes.

A minimalist API for C, Fortran, and Python enables to instrument the simulation code for the clients. A first call is required to connect the client to the server (*init_communication*). A *send* is issued to transfer time steps u_X^t as soon as computed. Eventually, a client calls *finialize_communication* to signal the server that no more data will be sent before disconnecting. We also provide a PDI plugin to interface with PDI instrumented simulation codes [68].

The launcher and server are developed in Python. Transport layer relies on ZMQ [32]. We are considering adding ADIOS2 [29] for gaining on data handling flexibility and better use high performance networks.

Regarding training, the framework supports the PyTorch and Tensorflow libraries. The training thread embeds a classical training loop where the main difference is the data source that relies on the training buffer instead of files. To ease the user transition from offline training for prototyping to online training, the buffer has been abstracted through the classical Tensorflow and Pytorch Dataset classes.

3.2 Data management

3.2.1 Data diversity. In classical offline training, the gradient descent expects batches built by uniformly sampling the fixed dataset, which can easily be done as the full dataset is available upfront [16]. The training process of our framework being online, it leads to inherent sources of data bias. Bias in the data is known to be detrimental to the quality of training. For instance, catastrophic forgetting characterizes network performance decrease on previously seen data when the training data distribution changes [39]. The sources of **workflow bias** caused by online training are of three different categories:

- *Inter-simulation*: The computational resources are finite. At any time, only c different clients can run simultaneously. The data produced by the c simulations may not present all the diversity the process f exhibits.
- *Intra-simulation*: Most of the time, the simulation codes executed by each client produce time steps in fixed increasing order. Each time step u_X^t is sent in the same order to the server as soon as computed. Obviously, time steps that have not been computed yet are not available for training.
- *Memory budget*: The training buffer has a fixed capacity n_c. At any time of the experiment, it can store only a subset of all the data received so far, $n_c \ll |\mathcal{X} \times \mathcal{T}|$ where $|\mathcal{X} \times \mathcal{T}|$ represents the cardinality of the set of data generated by the clients during the experiment.

The following details the different techniques the framework implements to ward off these biases and enable high-quality training and efficient use of resources.

3.2.2 Data distribution. The different GPUs involved in the training require the field u_X^t associated with each time step they receive to be complete. Each parallel client produces a time step partitioned across its different processes. The assembly of these parts is performed in situ on each client through an MPI gather on rank zero. Because training can usually be performed with data at a lower precision than the one produced by the solver, they are gathered and then converted, typically from 64 to 32 bits. Thus we avoid overloading the server with all these preprocessing tasks that need to be performed for each time step produced by each client.

To reduce inter and intra simulation bias, each client connects to all the ranks of the server and distributes the produced time steps u_X^t across all GPUs in a Round-Robin fashion. The destination of the first time step is chosen according to the client id to limit having all clients sending the same time step to the same GPU. This enforces the balance of data for data parallel training.

3.2.3 Training buffer. The training buffer is a fixed-size data container that has the dual role of accumulating a certain number of time steps to ensure batches contain well-mixed data to reduce workflow bias, and amortizing (up to a certain limit) discrepancies between data production and consumption to reduce resource idleness. The **Reservoir** algorithm we propose for managing this buffer is a key contribution of this paper. For comparison purposes, we also present two other strategies, FIFO and FIRO. Prior work only considered FIRO implementation [53], which is shown here to fail in optimizing GPU usage.

The classical **FIFO buffer (First In, First Out)** corresponds to the streaming case, where data are batched for training according to the order they are received. Each data produced is seen once, and only once, during training. Batch extraction is enabled as soon as the buffer can provide one. Compared to pure streaming, buffering provides some slack to keep the consumer, the learner thread, busy as long as batches are available in the buffer, even if data production is stopped or reduced for a while. Data production is suspended when the FIFO buffer is full. The FIFO buffer is thus simple to implement as a queue but does not enable mixing data beyond what the server receives.

The **FIRO buffer (First In, Random Out)** behaves very similarly to FIFO, with data evicted upon reading, except that these data are extracted from random positions to build less biased batches. Additionally, batches can only be extracted if the buffer is filled beyond a given *threshold*, again to reduce batch bias. The threshold

[1] https://gitlab.inria.fr/melissa/melissa

is set to zero once data production is over to enable consuming the last produced data. FIRO is implemented as a list container. Newly received samples are appended at its end.

Algorithm 1: Reservoir

```
Data: int capcity, int threshold, list seen, list not_seen, bool
      is_reception_over, lock lock
Function get():
    lock.acquire()
    while seen.size + not_seen.size ≤ threshold do
        wait  # Ensure there are enough data
    index = random.select(seen.size + not_seen.size)
    if index in not_seen.indices then
        item = not_seen[index]
        delete not_seen[index]
        if not is_reception_over then
            seen.append(item)
    else
        item = seen[index]
        if is_reception_over then
            delete seen[index] # Empty the buffer
    lock.release()
    return item
Function put(item):
    lock.acquire()
    while not_seen.size ≥ capacity do
        wait  # Block until one element gets seen
    if not_seen.size + seen.size ≥ capacity then
        index = random.select(seen.size)
        delete seen[index]  # Evict one seen element
    not_seen.append(item)
    lock.release()
```

The **Reservoir buffer** (Algorithm 1) enables data to be seen more than once to reduce consumer idleness in case of underproduction while giving priority to storing newly produced data over already seen ones. The Reservoir distinguishes the new unseen data from the ones already selected in a previous batch. When receiving new data while the buffer is full, a seen example is randomly evicted to make room for the incoming one. When building a batch, the elements are uniformly selected one by one among the seen and unseen elements in the buffer. Each selected unseen data is then moved with the seen ones. This Reservoir ensures that batches are well mixed; that data production can proceed as long as the buffer is not full of unseen samples, avoiding discarding any unseen data; and that data consumption is never locked, once the threshold is passed, as new batches can be built from already seen data. The Reservoir also has a *threshold* of minimum stored data before drawing batches. This ensures that the first received time steps are not over-represented in the first batches of the training. It also ensures the buffer has a minimum population to create diverse batches. When all the simulation data have been generated the blocking related to the threshold is lifted and the buffer state is updated. Batches can then be freely drawn from the buffer, regardless of its population size or the seen nature of the samples, until it finally empties out. When the reception is over and the buffer is empty, the training terminates. Notice that the amount of buffer space for unseen data is regulated by the incoming flow of new data, avoiding the static split that a dual buffer would require. Notice also that batch selection is with replacement. A selection without replacement would require a slight modification but would increase the selection cost. Experiments in Section 4 show the effectiveness of this training Reservoir.

For the Reservoir, data production pushes out data from the buffer. This can potentially lead to catastrophic forgetting as data from the earlier simulations get evicted. The expected residency time of a sample u_X^t in the buffer is about n_c, the buffer capacity or the number of unseen elements for the training buffer (Appendix A). Because of the online setting, the framework relies on the experimental design to ensure that the parameter space is well sampled to continuously populate the training buffer with diverse data. Experiments confirm that this avoids catastrophic forgetting. In Deep RL, a secondary replay buffer can be used to limit catastrophic forgetting, using the *Reservoir Sampling* algorithm [24, 90]. Reservoir sampling is a randomized algorithm to populate a k-size buffer from a stream of data guaranteeing that at any time τ the buffer is filled with k distinct elements uniformly sampled from the τ data received. Directly using this algorithm for online training would be counterproductive as it would waste the produced data not selected for inclusion in the buffer. As a secondary buffer, it would compete with the main buffer for the node memory.

4 EXPERIMENTS

Experiments[2] consist in training deep surrogates of a heat-equation solver (Section 4.1). We first assess the training throughput for the FIFO, FIRO and Reservoir buffers with a single GPU in Section 4.3, before comparing the obtained training quality in Section 4.4. Multi-GPU training is considered in Section 4.5. Section 4.6 finishes with a comparison between online and offline training at a larger scale.

4.1 Equation and deep surrogate architectures

The experiments consider the classical heat equation on a 2D rectangular domain (Equation 2):

$$\begin{cases} \frac{\partial T}{\partial t} = \alpha \nabla^2 T, \\ T(x,\, y,\, 0) = T_{\text{IC}}, \\ T(0,\, y,\, t) = T_{x_1},\; T(L,\, y,\, t) = T_{x_2}, \\ T(x,\, 0,\, t) = T_{y_1},\; T(x,\, L,\, t) = T_{y_2}, \end{cases} \tag{2}$$

where $T(x, y, t)$ denotes the field temperature, α the thermal diffusivity and $(T_{\text{IC}}, T_{x_1}, T_{y_1}, T_{x_2}, T_{y_2})$ the initial and 4 boundary temperatures. The solution is approximated with an in-house solver that implements a finite difference method with an implicit Euler scheme.

[2]Experiments are available for reproducibility at https://gitlab.inria.fr/melissa/sc2023

The temperature field $T(x, y, t)$ is discretized on a 1000×1000 Cartesian grid, and generated for 100 time steps representing $\Delta t = 0.01$ second each. The thermal diffusivity is fixed to $\alpha = 1$ m^2.s^{-1}. The solver code is written in Fortran90 and parallelized with MPI according to classical 2D domain partitioning.

The surrogate is trained to directly predict the temperature field $u^t_X = T(x, y, t)$ given the input (X, t), $X = (T_{\text{IC}}, T_{x_1}, T_{y_1}, T_{x_2}, T_{y_2})$. Training data are generated from solver runs taking as input 5 temperature parameters X (initial and boundary conditions) randomly sampled in $[100, 500]$K.

The deep surrogate architecture is a multilayer perceptron of 514M parameters (for reference the language model Bert-Large has about 340M parameters [22]), consisting of an input layer of 6 neurons, 2 hidden layers of 256 neurons with ReLU activation and an output of 1M neurons. It is trained using Adam optimizer with a starting learning rate of 1E^{-3}.

4.2 Computational resources

Experiments are performed on the Jean-Zay supercomputer ranked in the first half of the top500.org as of November 2022. The machine has one GPU and one CPU partition. Jean-Zay's CPU partition consists of 2 Intel Cascade Lake 6,248 processors with 20 cores at 2.5 GHz for a total of 40 cores per node. The GPU partition provides nodes accelerated with 4-GPU V100 32 GB. These GPU nodes came with 40 CPU and 160 GB of RAM. Jean-Zay's nodes are connected with Intel Omni-Path (100 GB/s). IBM Spectrum Scale (ex-GPFS) parallel file system with SSD disks (GridScaler GS18K SSD) manages the storage.

4.3 Throughput

The experiment illustrates how the different buffer implementations impact the throughput during training on 1 GPU. It considers a dataset of 250 runs of the heat equation solver, accounting for a total of 25,000 time steps. These data are generated by the framework running concurrently 100 clients on 50 nodes. Each instance of the solver executed by the clients is parallelized over 20 cores. Due to the limited support for heterogeneous jobs (CPU/GPU) on the machine, the framework adopts the submission process described before in Section 3.1. It submits three series of clients. First, it launches 100 simulations. Once the clients have terminated their executions, it launches the second series of 100 and finally the third series of the 50 remaining simulations. The throughput assesses the capacity of the framework to quickly provide training data to the GPU. The throughput is expressed as the number of samples per second processed by the GPU, computed on the learning thread over 10 successive batches every 10 batches. The batch size is fixed to 10 samples, with one sample being the time step u^t_X of one simulation associated with its 6 input parameters (X, t).

For this experiment only, as we are solely considering throughput, the performance of the trained network is never evaluated on a validation dataset. Performing the evaluation on the training thread would stall the data consumption of training batches, thus reducing throughput. Evaluation can either be excluded from the throughput measurement (as in the next experiments), or performed by using a dedicated GPU not involved in training.

The three training buffers described in Section 3.2.3, *FIFO*, *FIRO* and *Reservoir*, are evaluated. FIFO yields samples as they are received from the clients. FIRO and Reservoir have a fixed capacity of 6,000 samples (roughly a fourth of the whole simulated data) with a threshold set to 1,000 samples. These parameters remain the same in all the experiments.

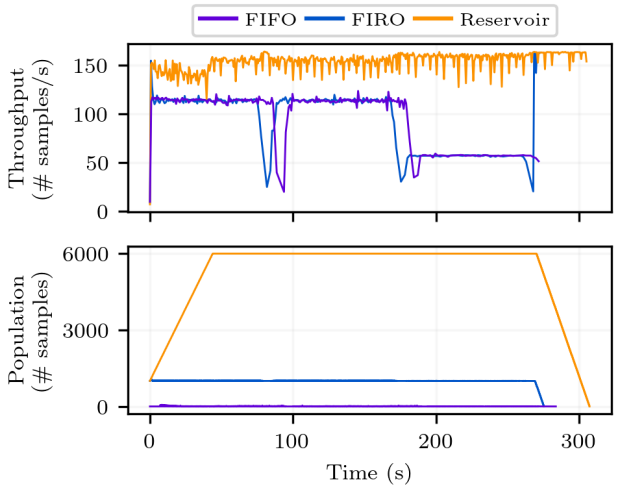

Figure 2: Reservoir population and throughput for different implementations. Each one handles a total of 25,000 time steps. The data are generated by successive series of 100, 100, and 50 clients running concurrently.

Figure 2 shows the evolution of the throughput with respect to time. The Reservoir provides the highest throughput of the three implementations. It appears that both FIFO and FIRO are sensitive to the flow of incoming data. Their throughput drops briefly at 100 and again at 200 seconds. After 200 seconds, their throughput is half what it was before. The times these drops occur coincide with the transitions between series of client submissions performed by the launcher. A bit before 100 seconds, all the running clients have terminated generating the data of the first 100 simulations. There is a delay before the clients start running the next 100 simulations and streaming the data they generate to the server. Similarly, there is a delay between the execution of the second series of 100 simulations and the execution of the last series of 50 ones. The amount of generated data is also halved during this last series running 50 instead of 100 concurrent simulations. Because FIFO yields data as they arrive these changes in data production result in drops in the throughput. The FIRO shows similar patterns, but it occurs sooner as the data consumption is stopped when reaching the threshold value and not when emptying the buffer as for FIFO.

Figure 2 bottom shows the FIFO and FIRO buffer populations stay around their minimum, respectively 0 and 1,000 set by the threshold value. Because data are consumed faster than produced their throughput is virtually the one of the data generation by the clients. At the end of the experiment the FIRO throughput rockets up. At this time, the data production by the clients has terminated and the blocking threshold has been released. Samples can thus be freely

drawn to form batches. The FIRO population, which is necessarily higher than the fixed threshold of 1,000 samples, can quickly yield 100 batches. Because the Reservoir evicts on writing rather than on reading like FIRO, its population increases quickly to its maximum, while FIRO's population remains around the threshold value, even when data consumption equals or overruns production as in this case. We can expect this enables Reservoir batches to present a higher diversity as they are sampled from a larger population.

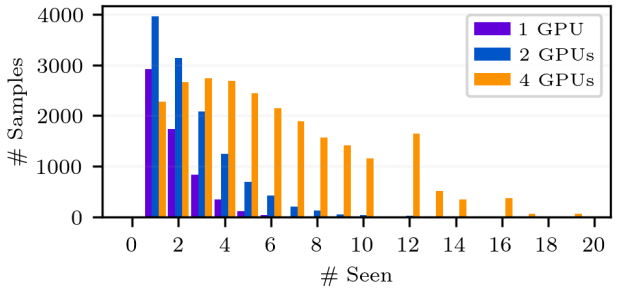

Figure 3: Number of occurrences of simulation time steps in batches for the Reservoir for different numbers of GPUs.

The Reservoir throughput remains constantly higher than the ones of FIFO and FIRO. The Reservoir manages to provide a high throughput by repeating samples, which erases the gaps between production and consumption rates. The visible high frequency fluctuations are not yet understood. Figure 3 presents a histogram of the number of times samples have been repeated for the Reservoir. Most of the samples have been seen in batches a couple of times, and at most, though rarely, 8 times during training. Thus, a few repetitions are enough to increase the throughput by 50%.

By providing higher throughput, the Reservoir maintains the GPU active, whereas for FIFO and FIRO implementations it can be idle, waiting for data to come. To be useful, this extra activity in training on more batches with already seen samples must result in increased generalization capabilities, which should appear as a lower validation loss.

4.4 Training quality

Figure 4 compares the training and validation losses for the different buffers. It also includes scores for offline training performed over one epoch with data read from files (data are seen only once). In all the different settings, the same unique time steps are seen during training. They only differ by how these time steps are ordered in training batches. For the same amount of unique data, offline training, whose batches are uniformly drawn from the full dataset, does not suffer the biases online methods experience. It is thus expected to provide the best achievable training quality and serve as a reference.

The training data generation replicates the process of the previous experiment (Section 4.3). The same validation dataset is used for all buffers. It consists of 10 simulations generated offline and never seen during training. Validation loss assesses the **generalization capabilities** achieved through training. Because validation occurs

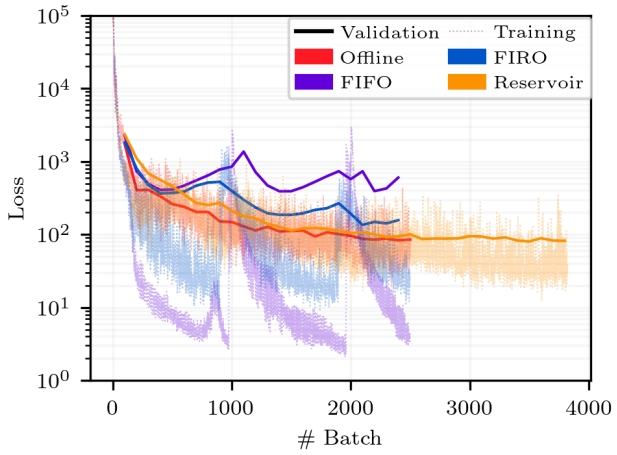

Figure 4: Comparison between training and validation losses for different buffer implementations.

on the training thread, it stalls the consumption of batches. Concurrently, incoming time steps are still being processed by the data aggregator thread to fill the buffer. As such, validation is a measure that impacts the experiment it measures. To mitigate this impact, validation is performed every 100 batches. During validation, new entries in the buffer are blocked by acquiring its mutex. Nonetheless, newly produced data sent by the clients still accumulate in the ZMQ buffer.

During training, the learning rate, initially set to $1E^{-3}$, is halved every 1000 batches. FIFO presents a low training loss associated with a high validation loss, which indicates overfitting. Both losses experience bursts, occurring when the learning rate is halved, which is symptomatic of unstable training. Although to a lesser extent, FIRO presents the same problems. On the contrary, Reservoir shows a more stable training not subject to overfitting. It achieves a validation loss that is on par with offline training. First, the performance of the FIFO simple data streaming confirms that online data generation indeed leads to a strong data bias that affects the training quality. Second, buffering with random data reads as performed by FIRO and Reservoir is effective in mitigating this bias. We assume the higher and thus more diverse population of Reservoir compared to FIRO (as shown in Figure 2) explains why this former implementation outperforms the latter in terms of generalization capabilities.

Because Reservoir can repeat samples, it can generate more batches than its counterparts, as it appears in Figure 4. Nonetheless, this higher number of training batches does not necessarily translate into longer training time. Table 1 indicates that training with Reservoir takes less than 6 minutes and is 45 seconds longer than with FIRO, while Offline training takes more than 1 hour to run.

4.5 Scaling to multiple GPUs

This experiment evaluates how the different training buffers behave when increasing the number of GPUs using data distributed parallel

training. The data generation, training process, and validation remain the same as in the previous experiment (Section 4.4). To keep the learning rate out of the impacting factors, its update frequency is scaled according to the number of GPUs to match always the same number of training samples. As previously, the learning rate is halved every 10,000 training samples until it reaches a minimum of $2.5E^{-4}$. Given a batch size of 10, these updates correspond to 1000, 500, and 250 batches for 1, 2, and 4 GPUs respectively.

Table 1 shows in its last column the average throughput during training for the different buffers and different numbers of GPUs. FIFO and FIRO fail to provide higher throughput when the number of GPUs increases. The population size displayed in Figure 2 was already showing FIFO and FIRO were unable to balance the production of data with the higher consumption. Increasing the number of GPUs only worsens this trend, because it increases the consumption and dilutes the generated data between the buffer replicas. Only the Reservoir scales with the number of GPUs at equal data production. It is important to remember however that because its higher throughput is due to data repetitions, this leads to more batches and thus training can eventually be slightly longer. Table 1 summarizes the results obtained for the different implementations and different numbers of GPUs.

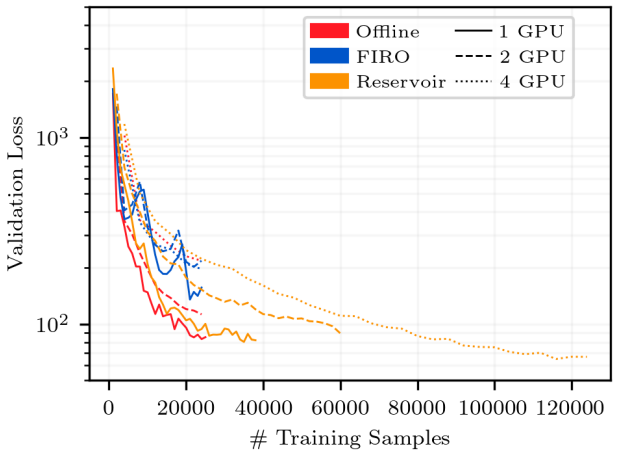

Figure 5: Comparison of the validation loss for different buffer implementations and number of GPUs. Training samples represent the number of simulation time steps, possibly with repetition, seen during training.

Figure 5 displays the validation loss when training with different numbers of GPUs. Beware that as the number of GPU increases the number of batches decreases, the data generation being always the same. To visually compare the different runs, the x-axis corresponds to the number of simulation time steps seen during training. This number, n_s, is related to the number of batches, n_b, by the relation $n_s = n_b \times b \times n_{\text{GPU}}$, where b is the batch size.

As in the previous experiment, the same number of unique samples is used for each training. Therefore, the generalization performance obtained with offline training, as an assumed optimal, still constitutes a benchmark. As the number of GPUs increases the validation loss decreases. This could be explained by the reduced number of optimization steps performed due to a lower number of batches.

The parameters of the buffer, namely the threshold and the maximum capacity, are the same for the different implementations and across the multiple GPUs. So the global buffer storage capacities increase with the number of GPUs, potentially enabling the production of more diverse batches. At an equivalent number of training samples, this greater diversity, however, does not compensate for the deterioration of the validation observed while training on more GPUs. Except for the run with 4 GPUs the validation loss obtained with FIRO is never matching the offline reference.

For Reservoir, increasing the number of GPUs also increases the global buffer size. But as the data production remains the same, the more GPUs, the less new data each local buffer receives. Besides, batches being larger with more GPUs, more data must be extracted at each training step. Thus, increasing the number of GPUs induces more sample repetition (as seen in Figure 3), which results in more training batches. There are 6 times more batches generated by Reservoir with 4 GPUs compared to 1. These additional batches trigger more optimization steps. The greater number of optimization steps that characterize Reservoir can explain why it beats the one-epoch offline training benchmark, not unlike how training for multiple epochs would improve the offline setting. However, Reservoir repeats samples at a much faster pace than would an offline multi-epoch training (Table 1).

Reservoir consistently outperforms all the other training settings for the same number of GPUs. At the end of each training, the validation loss of Reservoir is significantly lower than the one of FIRO, often more than halved. The Reservoir capability to keep its buffer full with a self-adjusted amount of new and already-seen data makes it more amenable to take full benefit of multi-GPU training compared to FIRO. It provides higher throughput and better generalization materialized in a lower validation loss.

4.6 Online versus multi-epoch Offline

We have seen that properly managed online training, with Reservoir being the best option, is already competitive with offline training on a single epoch. But the true potential of online training comes when enabling working with datasets so large that they cannot be reasonably handled in an offline fashion, due to storage and I/O costs. Offline needs to restrain to a reasonable dataset presented several times through different epochs, while online can lean towards training on a potentially unlimited dataset. In the following, we compare offline and online in such a context.

Offline training is performed on the same 250 simulations data as used for previous experiments, except that here we perform 100 epochs (Figure 6). The dataset is 450 GB raw with one file per time step, 95.5 GB when compressed using one binary file per simulation. The data generation is also performed in parallel with the framework using 2000 cores, but instead of streaming the data through the API, they are simply written on disk. Here, the framework reveals itself also useful to quickly generate datasets by leveraging the parallelism of its clients. During training, the data are loaded from SSD disk. The loading relies on *mmap* to read only the requested time sep without having to load the entire file in memory.

Table 1: Comparison of the training and throughput performances for different numbers of GPUs. All experiments rely on 250 simulations producing 100 GB of data or 25,000 unique samples. Fifty nodes are pre-allocated to run 100 simultaneous simulations, each one using 20 cores. The *Min. RMSE* column indicates the validation loss obtained after training.

Buffer	Gpu number (n)	Generation (hours)	Total (hours)	Min. MSE ↓	Mean. Throughput (samples/sec)
Offline	1	0.22	1.13	83.1	13.2
FIFO	1	—	0.0805	391	118
FIRO	1	—	0.0832	135	114
Reservoir	1	—	0.0928	80.3	147.6
Offline	2	0.22	0.353	112	30.2
FIFO	2	—	**0.0793**	384	105
FIRO	2	—	0.0835	202	98.1
Reservoir	2	—	0.0972	89.3	212
Offline	4	0.22	0.201	218	43.2
FIFO	4	—	0.0799	445	100
FIRO	4	—	0.0824	197	96.4
Reservoir	4	—	0.0952	**65.0**	**476**

Table 2: Comparison with 4 GPUs. The *RESOURCES* column indicates computing resources used for data generation and training. The online experiment trains the network with 20,000 simulations producing 8TB of data or 2,000,000 unique samples. A total of 128 nodes are pre-allocated to run 512 simultaneous simulations, each one using 10 cores.

Buffer	Generation/Training Resources(Cores & Gpu)	Generation (hours)	Total (hours)	Dataset (GB)	Unique Samples (*N*)	MSE ↓	Throughput (samples/sec)
Offline	2,000C / 40C, 4G	0.22	24.5	100	25,000	25.1	38.2
Reservoir	5,120C / 40C, 4G	—	1.97	8,000	2,000,000	13.2	476.7

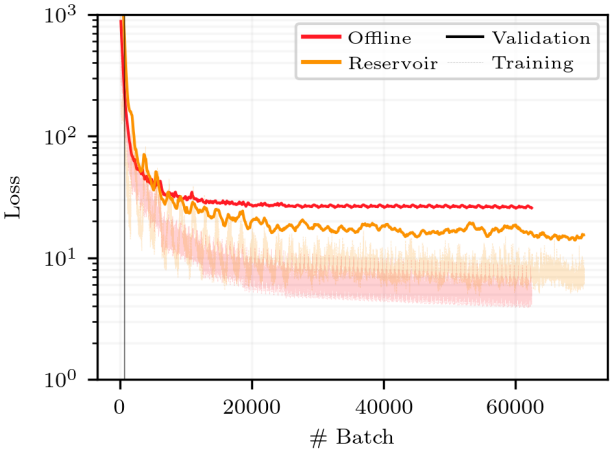

Figure 6: Comparison between offline and online training at equivalent numbers of batches. Offline trains for several epochs on the data generated by 250 simulations. The black vertical line corresponds to the first epoch. Online training is performed on 20,000 simulations managed by the framework.

On each of the 4 GPUs synchronized with Pytorch Distributed and individually associated to 10 cores, the Dataloader retrieves batches with 8 parallel workers. Although more advanced techniques exist for fast data loading of massive deep learning datasets [4, 23, 60, 92], we believe the current process is fairly representative of common practice in the deep learning community.

Online training is performed with 20,000 simulations managed by the framework. The simulation clients run on 5,120 cores and generate a total of 8TB that are forwarded as soon as produced to the training server running with 4 GPUs.

Figure 6 compares the training and validation losses for the two settings. The offline training displays clear signs of overfitting with a validation loss that has converged to a higher value than the training loss that keeps decreasing. Overfitting is less pronounced with the Reservoir as both validation and training losses keep decreasing. Overall, Reservoir training significantly improves the validation loss, and so the surrogate generalization capabilities, by 47%.

Table 2 summarizes the parameters of these two training settings, including throughput and execution times. Using both 4 GPUs, offline only manage to process about 38 samples/sec, even when using 8 data loaders per GPU, incurring a training of 24h. Online Reservoir training enables to process 476 samples/sec, leading to a combined data generation and training in less than 2h.

5 CONCLUSION

In this paper, we presented the Melissa framework and introduced the Reservoir algorithm. The former enables online training from an ensemble of simulation runs. The latter is a training buffer that

adequately mitigates the bias inherent to online learning while optimizing throughput. The combination of both allows high-quality training of deep surrogates. Experiments revealed that GPUs can indeed process batches at high frequency, way beyond what is achieved with offline training. Using consolidated figures provided by the supercomputer center (1 kh/core CPU = 6€, 1 kh/GPU V100 = 360€, 1TB (SSD storage) = 56€), leads to a cost of our large experiment (Table 2) with online training at 63.8€, only 29% above the cost of offline data generation and training at 49.1€. The cost of offline training would decrease to 41.16€ when repeated (no storage and data generation costs). If offline training would have been performed with the 8TB dataset of online training, the sole storage cost would account for 480€. A realistic production workflow will likely combine pre-training (with the necessary repetitions to tune hyperparameters) from a static reduced dataset and few online re-training at scale with complementary data so as to reach the best possible generalization capabilities. This will enable to control the trade-off between storage footprint and the computing cost of re-running simulations for each new training.

The present work does not include the use of the surrogate. Thus, the global gain, counting the cost of training and use of the surrogate, versus using only the original solver cannot be evaluated. But higher generalization capabilities mean a surrogate capable of giving higher quality results, likely leading to reduce surrogate runs as well as some of the simulation runs that are often required to bring higher quality data into the process.

Experiments conducted for this paper (including preparatory and test runs) account for 584062 core.h and 4770 gpu.h, leading to 1.1TCO2e emissions (counting direct energy use, operating costs and hardware construction), or 44% of the one person round-trip flight from Paris to Denver in economy class that presenting this paper will require.

The framework can support adaptive training where the next set of clients to run is defined online according to the current training status. This could increase generalization capabilities while requiring fewer simulations to run. It is only possible in the online context the framework provides. This will be the object of future investigations.

ACKNOWLEDGMENTS

This work was performed using HPC/AI resources from GENCI-IDRIS (Grant 2022-[AD010610366R1]), and received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 956560.

A RESERVOIR

Proof. Let's consider a container of fixed capacity n on which m new items are sequentially added, with $m \gg n$. New items are inserted at random locations, overwriting any data already present in the container at these locations. Considering a new insertion, the probability for an already present item to remain in the container is the probability to select any location other than the one where the item lies *i.e* $\frac{n-1}{n}$. Thus, for an item to remain in the buffer after k insertions the probability is defined by Equation 3. The factor $\frac{1}{n}$ comes from normalization so to have $\sum_{k=0}^{+\infty} p(k) = 1$.

$$p(k) = \frac{1}{n}(1 - \frac{1}{n})^k \tag{3}$$

The expected residency time τ in the container for any item is then given by $\sum_{k=0}^{+\infty} k \cdot p(k)$.

$$\tau = \frac{1}{n} \sum_{k=0}^{+\infty} k(1 - \frac{1}{n})^k \tag{4}$$

$$= n - 1 \tag{5}$$

The step from Equation 4 to Equation 5 is made by recognizing in Equation 4 the derivative of a converging geometric series.

To intuitively understand this result, one can consider, as the insertion is done at a random location, it is as if the eviction of old items is performed sequentially. Hence, an expected residency time for any item of $n - 1$. □

REFERENCES

[1] Georges Aad, Brad Abbott, Dale C Abbott, A Abed Abud, Kira Abeling, Deshan Kavishka Abhayasinghe, Syed Haider Abidi, Asmaa Aboulhorma, Halina Abramowicz, Henso Abreu, et al. 2022. AtlFast3: the next generation of fast simulation in ATLAS. *Computing and software for big science* 6, 1 (2022), 7.

[2] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C Smith, Berk Hess, and Erik Lindahl. 2015. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX* 1 (2015), 19–25.

[3] Dong H Ahn, Ned Bass, Albert Chu, Jim Garlick, Mark Grondona, Stephen Herbein, Helgi I Ingólfsson, Joseph Koning, Tapasya Patki, Thomas RW Scogland, et al. 2020. Flux: Overcoming scheduling challenges for exascale workflows. *Future Generation Computer Systems* 110 (2020), 202–213.

[4] Alex Aizman, Gavin Maltby, and Thomas Breuel. 2019. High Performance I/O For Large Scale Deep Learning. In *2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, December 9-12, 2019*. IEEE, 5965–5967. https://doi.org/10.1109/BigData47090.2019.9005703

[5] Francis J Alexander, James Ang, Jenna A Bilbrey, Jan Balewski, Tiernan Casey, Ryan Chard, Jong Choi, Sutanay Choudhury, Bert Debusschere, Anthony M DeGennaro, Nikoli Dryden, J Austin Ellis, Ian Foster, Cristina Garcia Cardona, Sayan Ghosh, Peter Harrington, Yunzhi Huang, Shantenu Jha, Travis Johnston, Ai Kagawa, Ramakrishnan Kannan, Neeraj Kumar, Zhengchun Liu, Naoya Maruyama, Satoshi Matsuoka, Erin McCarthy, Jamaludin Mohd-Yusof, Peter Nugent, Yosuke Oyama, Thomas Proffen, David Pugmire, Sivasankaran Rajamanickam, Vinay Ramakrishniah, Malachi Schram, Sudip K Seal, Ganesh Sivaraman, Christine Sweeney, Li Tan, Rajeev Thakur, Brian Van Essen, Logan Ward, Paul Welch, Michael Wolf, Sotiris S Xantheas, Kevin G Yager, Shinjae Yoo, and Byung-Jun Yoon. 2021. Co-design Center for Exascale Machine Learning Technologies (ExaLearn). *The International Journal of High Performance Computing Applications* 35, 6 (2021), 598–616. https://doi.org/10.1177/10943420211029302 Publisher: SAGE Publications Ltd STM.

[6] Kelsey R Allen, Tatiana Lopez-Guavara, Kim Stachenfeld, Alvaro Sanchez-Gonzalez, Peter Battaglia, Jessica B Hamrick, and Tobias Pfaff. 2022. Inverse Design for Fluid-Structure Interactions using Graph Network Simulators. In *Advances in Neural Information Processing Systems*, Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.). https://openreview.net/forum?id=HaZuqj0Gvp2

[7] Jeffrey Anderson, Tim Hoar, Kevin Raeder, Hui Liu, Nancy Collins, Ryan Torn, and Avelino Avellano. 2009. The Data Assimilation Research Testbed: A Community Facility. *Bulletin of the American Meteorological Society* 90, 9 (2009), 1283–1296. Publisher: American Meteorological Society.

[8] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Luksaz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. 2019. Parsl: Pervasive Parallel Programming in Python. In *28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 25–36.

[9] Vivek Balasubramanian, Travis Jensen, Matteo Turilli, Peter Kasson, Michael Shirts, and Shantenu Jha. 2020. Adaptive Ensemble Biomolecular Applications at Scale. *SN Computer Science* 1, 2 (2020), 1–15.

[10] Vivek Balasubramanian, Shantenu Jha, Andre Merzky, and Matteo Turilli. 2019. RADICAL-Cybertools: Middleware Building Blocks for Scalable Science. https://arxiv.org/abs/1904.03085

[11] Vivek Balasubramanian, Matteo Turilli, Weiming Hu, Matthieu Lefebvre, Wenjie Lei, Guido Cervone, Jeroen Tromp, and Shantenu Jha. 2018. Harnessing the

Power of Many: Extensible Toolkit for Scalable Ensemble Applications. In *IPDPS 2018*. 536–545.

[12] Andrew C Bauer, Hasan Abbasi, James Ahrens, Hank Childs, Berk Geveci, Scott Klasky, Kenneth Moreland, Patrick O'Leary, Venkatram Vishwanath, Brad Whitlock, et al. 2016. In situ methods, infrastructures, and applications on high performance computing platforms. In *Computer Graphics Forum*, Vol. 35. Wiley Online Library, 577–597.

[13] Atilim Güneş Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, Lawrence Meadows, Jialin Liu, Andreas Munk, Saeid Naderiparizi, Bradley Gram-Hansen, Gilles Louppe, et al. 2019. Etalumis: Bringing probabilistic programming to scientific simulators at scale. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–24.

[14] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. 2019. Dota 2 with large scale deep reinforcement learning. *ArXiv preprint* abs/1912.06680 (2019). https://arxiv.org/abs/1912.06680

[15] Bartosz Bosak, Tomasz Piontek, Paul Karlshoefer, Erwan Raffin, Jalal Lakhlili, and Piotr Kopta. 2021. Verification, Validation and Uncertainty Quantification of Large-Scale Applications with QCG-PilotJob. In *Computational Science – ICCS 2021 (Lecture Notes in Computer Science)*, Maciej Paszynski, Dieter Kranzlmüller, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M.A. Sloot (Eds.). Springer International Publishing, Cham, 495–501. https://doi.org/10.1007/978-3-030-77977-1_39

[16] Léon Bottou, Frank E Curtis, and Jorge Nocedal. 2018. Optimization methods for large-scale machine learning. *Siam Review* 60, 2 (2018), 223–311.

[17] Alexander Brace, Igor Yakushin, Heng Ma, Anda Trifan, Todd Munson, Ian Foster, Arvind Ramanathan, Hyungro Lee, Matteo Turilli, and Shantenu Jha. 2022. Coupling streaming AI and HPC ensembles to achieve 100–1000× faster biomolecular simulations. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 806–816.

[18] Johannes Brandstetter, Daniel E. Worrall, and Max Welling. 2022. Message Passing Neural PDE Solvers. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net. https://openreview.net/forum?id=vSix3HPYKSU

[19] Steven L Brunton, Bernd R Noack, and Petros Koumoutsakos. 2020. Machine learning for fluid mechanics. *Annual review of fluid mechanics* 52 (2020), 477–508.

[20] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard. 2005. A batch scheduler with high level components. In *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, Vol. 2. IEEE, 776–783.

[21] Kyle Cranmer, Johann Brehmer, and Gilles Louppe. 2020. The frontier of simulation-based inference. *Proceedings of the National Academy of Sciences* 117, 48 (2020), 30055–30062.

[22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[23] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. 2021. Clairvoyant prefetching for distributed machine learning I/O. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[24] Pavlos S Efraimidis and Paul G Spirakis. 2006. Weighted random sampling with a reservoir. *Information processing letters* 97, 5 (2006), 181–185.

[25] Wael R Elwasif, David E Bernholdt, Sreekanth Pannala, Srikanth Allu, and Samantha S Foley. 2012. Parameter sweep and optimization of loosely coupled simulations using the DAKOTA toolkit. In *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*. 102–110.

[26] William Fedus, Prajit Ramachandran, Rishabh Agarwal, Yoshua Bengio, Hugo Larochelle, Mark Rowland, and Will Dabney. 2020. Revisiting Fundamentals of Experience Replay. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 3061–3071. http://proceedings.mlr.press/v119/fedus20a.html

[27] Sebastian Friedemann and Bruno Raffin. 2022. An elastic framework for ensemble-based large-scale data assimilation. *The international journal of high performance computing applications* 36, 4 (2022), 543–563.

[28] Kai Fukami, Koji Fukagata, and Kunihiko Taira. 2021. Machine-learning-based spatio-temporal super resolution reconstruction of turbulent flows. *Journal of Fluid Mechanics* 909 (2021), A9.

[29] William F Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, et al. 2020. Adios 2: The adaptable input output system. a framework for high-performance data management. *SoftwareX* 12 (2020), 100561.

[30] Priya Goyal, Piotr Dollár, 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* abs/1706.02677 (2017). https://arxiv.org/abs/1706.02677

[31] Oliver Hennigh, Susheela Narasimhan, Mohammad Amin Nabian, Akshay Subramaniam, Kaustubh Tangsali, Zhiwei Fang, Max Rietmann, Wonmin Byeon, and Sanjay Choudhry. 2021. NVIDIA SimNet™: An AI-accelerated multi-physics simulation framework. In *International Conference on Computational Science*. Springer, 447–461.

[32] Pieter Hintjens. 2013. *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.".

[33] Steven CH Hoi, Doyen Sahoo, Jing Lu, and Peilin Zhao. 2021. Online learning: A comprehensive survey. *Neurocomputing* 459 (2021), 249–289.

[34] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. 2018. Distributed Prioritized Experience Replay. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=H1Dy---0Z

[35] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*, Ameet Talwalkar, Virginia Smith, and Matei Zaharia (Eds.). mlsys.org. https://proceedings.mlsys.org/book/265.pdf

[36] George Em Karniadakis, Ioannis G Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. 2021. Physics-informed machine learning. *Nature Reviews Physics* 3, 6 (2021), 422–440.

[37] Muhammad Firmansyah Kasim, D Watson-Parris, L Deaconu, S Oliver, P Hatfield, Dustin H Froula, Gianluca Gregori, M Jarvis, S Khatiwala, J Korenaga, et al. 2021. Building high accuracy emulators for scientific simulations with deep neural architecture search. *Machine Learning: Science and Technology* 3, 1 (2021), 015013.

[38] Byungsoo Kim, Vinicius C. Azevedo, Nils Thuerey, Theodore Kim, Markus Gross, and Barbara Solenthaler. 2019. Deep Fluids: A Generative Network for Parameterized Fluid Simulations. *Computer Graphics Forum (Proc. Eurographics)* 38, 2 (2019).

[39] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences* 114, 13 (2017), 3521–3526.

[40] Dmitrii Kochkov, Jamie A Smith, Ayya Alieva, Qing Wang, Michael P Brenner, and Stephan Hoyer. 2021. Machine learning–accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences* 118, 21 (2021).

[41] Aditi S. Krishnapriyan, Amir Gholami, Shandian Zhe, Robert M. Kirby, and Michael W. Mahoney. 2021. Characterizing possible failure modes in physics-informed neural networks. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (Eds.). 26548–26560. https://proceedings.neurips.cc/paper/2021/hash/df438e5206f31600e6ae4af72f2725f1-Abstract.html

[42] Alexander Lavin, David Krakauer, Hector Zenil, Justin Gottschlich, Tim Mattson, Johann Brehmer, Anima Anandkumar, Sanjay Choudry, Kamil Rocki, Atılım Güneş Baydin, Carina Prunkl, Brooks Paige, Olexandr Isayev, Erik Peterson, Peter L. McMahon, Jakob Macke, Kyle Cranmer, Jiaxin Zhang, Haruko Wainwright, Adi Hanuka, Manuela Veloso, Samuel Assefa, Stephan Zheng, and Avi Pfeffer. 2021. Simulation Intelligence: Towards a New Generation of Scientific Methods. https://arxiv.org/abs/2112.03235

[43] Hyungro Lee, Andre Merzky, Li Tan, Mikhail Titov, Matteo Turilli, Dario Alfe, Agastya Bhati, Alex Brace, Austin Clyde, Peter Coveney, et al. 2021. Scalable HPC & AI infrastructure for COVID-19 therapeutics. In *Proceedings of the Platform for Advanced Scientific Computing Conference*. 1–13.

[44] Hyungro Lee, Matteo Turilli, Shantenu Jha, Debsindhu Bhowmik, Heng Ma, and Arvind Ramanathan. 2019. Deepdrivemd: Deep-learning driven adaptive molecular simulations for protein folding. In *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE, 12–19.

[45] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proceedings of the VLDB Endowment* 13, 12 (2020).

[46] Zongyi Li, Nikola Borislavov Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew M. Stuart, and Anima Anandkumar. 2021. Fourier Neural Operator for Parametric Partial Differential Equations. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=c8P9NQVtmnO

[47] Zijie Li, Kazem Meidani, and Amir Barati Farimani. 2023. Transformer for Partial Differential Equations' Operator Learning. *Transactions on Machine Learning Research* (2023). https://openreview.net/forum?id=EPPqt3uERT

[48] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael I. Jordan, and Ion Stoica. 2018. RLlib: Abstractions for Distributed Reinforcement Learning. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan,*

Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research, Vol. 80), Jennifer G. Dy and Andreas Krause (Eds.). PMLR, 3059–3068. http://proceedings.mlr.press/v80/liang18b.html

[49] Didier Lucor, Atul Agrawal, and Anne Sergent. 2022. Simple computational strategies for more effective physics-informed neural networks modeling of turbulent natural convection. *J. Comput. Phys.* 456 (2022), 111022.

[50] Sergio M. Martin, Daniel Wälchli, Georgios Arampatzis, Athena E. Economides, Petr Karnakov, and Petros Koumoutsakos. 2022. Korali: Efficient and scalable software framework for Bayesian uncertainty quantification and stochastic optimization. *Computer Methods in Applied Mechanics and Engineering* 389 (2022), 114264. https://doi.org/10.1016/j.cma.2021.114264

[51] Romit Maulik, Kai Fukami, Nesar Ramachandra, Koji Fukagata, and Kunihiko Taira. 2020. Probabilistic neural networks for fluid flow surrogate modeling and data recovery. *Physical Review Fluids* 5, 10 (2020), 104401.

[52] Andre Merzky, Mark Santcroos, Matteo Turilli, and Shantenu Jha. 2015. Radical-pilot: Scalable execution of heterogeneous and dynamic workloads on supercomputers. *CoRR, abs/1512.08194* (2015).

[53] Lucas Thibaut Meyer, Marc Schouler, Robert Alexander Caulk, Alejandro Ribes, and Bruno Raffin. 2023. Training Deep Surrogate Models with Large Scale Online Learning. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 24614–24630. https://proceedings.mlr.press/v202/meyer23b.html

[54] Hiroaki Mikami, Hisahiro Suganuma, Pongsakorn U.-Chupala, Yoshiki Tanaka, and Yuichi Kageyama. 2018. ImageNet/ResNet-50 Training in 224 Seconds. *ArXiv preprint* abs/1811.05233 (2018). https://arxiv.org/abs/1811.05233

[55] Parviz Moin and Krishnan Mahesh. 1998. Direct numerical simulation: a tool in turbulence research. *Annual review of fluid mechanics* 30, 1 (1998), 539–578.

[56] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: a distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Carlsbad, CA, USA, 561–577.

[57] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. 2015. Massively parallel methods for deep reinforcement learning. *ArXiv preprint* abs/1507.04296 (2015). https://arxiv.org/abs/1507.04296

[58] Lars Nerger and Wolfgang Hiller. 2013. Software for ensemble-based data assimilation systems—Implementation strategies and scalability. *Computers & Geosciences* 55 (2013), 110–118.

[59] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. 2019. Continual lifelong learning with neural networks: A review. *Neural Networks* 113 (2019), 54–71.

[60] Arnab K. Paul, Ahmad Maroof Karimi, and Feiyi Wang. 2021. Characterizing Machine Learning I/O Workloads on Leadership Scale HPC Systems. In *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 1–8. https://doi.org/10.1109/MASCOTS53633.2021.9614303

[61] Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W. Battaglia. 2021. Learning Mesh-Based Simulation with Graph Networks. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=roNqYL0_XP

[62] S. Pronk, G. R. Bowman, B. Hess, P. Larsson, I. S. Haque, V. S. Pande, I. Pouya, K. Beauchamp, P. M. Kasson, and E. Lindahl. 2011. Copernicus: A new paradigm for parallel adaptive molecular dynamics. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–10. https://doi.org/10.1145/2063384.2063465

[63] Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, Ali Ramadhan, and Alan Edelman. 2020. Universal differential equations for scientific machine learning. *ArXiv preprint* abs/2001.04385 (2020). https://arxiv.org/abs/2001.04385

[64] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. 2019. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics* 378 (2019), 686–707.

[65] Martin Riedmiller, Jost Tobias Springenberg, Roland Hafner, and Nicolas Heess. 2022. Collect & infer-a fresh look at data-efficient reinforcement learning. In *Conference on Robot Learning*. PMLR, 1736–1744.

[66] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. *Proceedings of the 14th Python in Science Conference* (2015), 126–132. https://doi.org/10.25080/Majora-7b98e3ed-013 Conference Name: Proceedings of the 14th Python in Science Conference.

[67] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*. Springer, 234–241.

[68] Corentin Roussel, Kai Keller, Mohamed Gaalich, Leonardo Bautista Gomez, and Julien Bigot. 2017. PDI, an approach to decouple I/O concerns from high-performance simulation codes. (2017). https://hal.archives-ouvertes.fr/hal-01587075 working paper or preprint.

[69] Doyen Sahoo, Quang Pham, Jing Lu, and Steven C. H. Hoi. 2018. Online Deep Learning: Learning Deep Neural Networks on the Fly. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, Jérôme Lang (Ed.). ijcai.org, 2660–2666. https://doi.org/10.24963/ijcai.2018/369

[70] Victor Garcia Satorras, Emiel Hoogeboom, and Max Welling. 2021. E(n) Equivariant Graph Neural Networks. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 9323–9332. http://proceedings.mlr.press/v139/satorras21a.html

[71] Marc Schouler, Robert Alexander Caulk, Lucas Meyer, Théophile Terraz, Christoph Conrads, Sebastian Friedemann, Achal Agarwal, Juan Manuel Baldonado, Bartłomiej Pogodziński, Anna Sekuła, Alejandro Ribes, and Bruno Raffin. 2023. Melissa: coordinating large-scale ensemble runs for deep learning and sensitivity analyses. *Journal of Open Source Software* 8, 86 (2023), 5291. https://doi.org/10.21105/joss.05291

[72] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *ArXiv preprint* abs/1909.08053 (2019). https://arxiv.org/abs/1909.08053

[73] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. 2017. Mastering the game of Go without human knowledge. *Nature* 550, 7676 (2017), 354–359. https://doi.org/10.1038/nature24270

[74] Justin Sirignano and Konstantinos Spiliopoulos. 2018. DGM: A deep learning algorithm for solving partial differential equations. *Journal of computational physics* 375 (2018), 1339–1364.

[75] Rick Stevens, Valerie Taylor, Jeff Nichols, Arthur Barney Maccabe, Katherine Yelick, and David Brown. 2020. *AI for Science: Report on the Department of Energy (DOE) Town Halls on Artificial Intelligence (AI) for Science.* Technical Report. Argonne National Lab.(ANL), Argonne, IL (United States).

[76] Makoto Takamoto, Timothy Praditia, Raphael Leiteritz, Daniel MacKinlay, Francesco Alesiani, Dirk Pflüger, and Mathias Niepert. 2022. PDEBench: An extensive benchmark for scientific machine learning. *Advances in Neural Information Processing Systems* 35 (2022), 1596–1611.

[77] Meng Tang, Yimin Liu, and Louis J Durlofsky. 2020. A deep-learning-based surrogate model for data assimilation in dynamic subsurface flow problems. *J. Comput. Phys.* 413 (2020), 109456.

[78] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M Badia, Jordi Torres, Toni Cortes, and Jesús Labarta. 2017. PyCOMPSs: Parallel computational workflows in Python. *The International Journal of High Performance Computing Applications* 31, 1 (2017), 66–82. https://doi.org/10.1177/1094342015594678 Publisher: SAGE Publications Ltd STM.

[79] Théophile Terraz, Alejandro Ribes, Yvan Fournier, Bertrand Iooss, and Bruno Raffin. 2017. Melissa: large scale in transit sensitivity analysis avoiding intermediate files. In *Proceedings of the international conference for high performance computing, networking, storage and analysis (SC'17)*. 1–14.

[80] Nils Wandel, Michael Weinmann, and Reinhard Klein. 2021. Learning Incompressible Fluid Dynamics from Scratch - Towards Fast, Differentiable Fluid Models that Generalize. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=KUDUoRsEphu

[81] Rui Wang, Karthik Kashinath, Mustafa Mustafa, Adrian Albert, and Rose Yu. 2020. Towards Physics-informed Deep Learning for Turbulent Flow Prediction. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash (Eds.). ACM, 1457–1466. https://dl.acm.org/doi/10.1145/3394486.3403198

[82] Shangying Wang, Kai Fan, Nan Luo, Yangxiaolu Cao, Feilun Wu, Carolyn Zhang, Katherine A Heller, and Lingchong You. 2019. Massive computational acceleration by using neural networks to emulate mechanism-based biological models. *Nature communications* 10, 1 (2019), 1–9.

[83] Logan Ward, Ganesh Sivaraman, J Gregory Pauloski, Yadu Babuji, Ryan Chard, Naveen Dandu, Paul C Redfern, Rajeev S Assary, Kyle Chard, Larry A Curtiss, et al. 2021. Colmena: Scalable machine-learning-based steering of ensemble simulations for high performance computing. In *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*. IEEE, 9–20.

[84] Jiayi Weng, Min Lin, Shengyi Huang, Bo Liu, Denys Makoviichuk, Viktor Makoviychuk, Zichen Liu, Yufan Song, Ting Luo, Yukun Jiang, et al. 2022. Envpool: A highly parallel reinforcement learning environment execution engine. *Advances in Neural Information Processing Systems* 35 (2022), 22409–22421.

[85] An Xu, Zhouyuan Huo, and Heng Huang. 2020. On the Acceleration of Deep Learning Model Parallelism With Staleness. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*. IEEE, 2085–2094. https://doi.org/10.1109/CVPR42600.2020.00216

[86] H. Yashiro, K. Terasaki, Y. Kawai, S. Kudo, T. Miyoshi, T. Imamura, K. Minami, H. Inoue, T. Nishiki, T. Saji, M. Satoh, and H. Tomita. 2020. A 1024-Member Ensemble Data Assimilation with 3.5-Km Mesh Global Weather Simulations. In *Supercomputing 2020: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–10.

[87] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*. Springer, 44–60.

[88] Yuliana Zamora, Logan Ward, Ganesh Sivaraman, Ian Foster, and Henry Hoffmann. 2021. Proxima: accelerating the integration of machine learning in atomistic simulations. In *Proceedings of the ACM International Conference on Supercomputing (ICS '21)*. Association for Computing Machinery, New York, NY, USA, 242–253. https://doi.org/10.1145/3447818.3460370

[89] Daochen Zha, Kwei-Herng Lai, Kaixiong Zhou, and Xia Hu. 2019. Experience Replay Optimization. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, Sarit Kraus (Ed.). ijcai.org, 4243–4249. https://doi.org/10.24963/ijcai.2019/589

[90] Linjing Zhang, Zongzhang Zhang, Zhiyuan Pan, Yingfeng Chen, Jiangcheng Zhu, Zhaorong Wang, Meng Wang, and Changjie Fan. 2019. A framework of dual replay buffer: balancing forgetting and generalization in reinforcement learning. In *Proceedings of the 2nd Workshop on Scaling Up Reinforcement Learning (SURL), International Joint Conference on Artificial Intelligence (IJCAI)*.

[91] Yinhao Zhu, Nicholas Zabaras, Phaedon-Stelios Koutsourelakis, and Paris Perdikaris. 2019. Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data. *J. Comput. Phys.* 394 (2019), 56–81.

[92] Zongwei Zhu, Luchao Tan, Yinzhen Li, and Cheng Ji. 2020. PHDFS: Optimizing I/O Performance of HDFS in Deep Learning Cloud Computing Platform. *Journal of Systems Architecture* 109 (2020), 101810. https://doi.org/10.1016/j.sysarc.2020.101810