

Deep Generative Models that Solve PDEs: Distributed Computing for Training Large Data-Free Models

Sergio Botelho[§]
RocketML Inc.
sergio@rocketml.net

Ameya Joshi[§]
New York University (NYU)
ameya.joshi@nyu.edu

Biswajit Khara[§]
Iowa State University
bkhara@iastate.edu

Vinay Rao
RocketML Inc.
vinay@rocketml.net

Soumik Sarkar
Iowa State University
soumiks@iastate.edu

Chinmay Hegde
New York University (NYU)
chinmay.h@nyu.edu

Santi Adavani
RocketML Inc.
santi@rocketml.net

Baskar Ganapathysubramanian
Iowa State University
baskarg@iastate.edu

Abstract—Recent progress in scientific machine learning (SciML) has opened up the possibility of training novel neural network architectures that solve complex partial differential equations (PDEs). Several (nearly data free) approaches have been recently reported that successfully solve PDEs, with examples including deep feed forward networks, generative networks, and deep encoder-decoder networks. However, practical adoption of these approaches is limited by the difficulty in training these models, especially to make predictions at large output resolutions ($\geq 1024 \times 1024$).

Here we report on a software framework for data parallel distributed deep learning that resolves the twin challenges of training these large SciML models training in reasonable time as well as distributing the storage requirements. Our framework provides several out of the box functionality including (a) loss integrity independent of number of processes, (b) synchronized batch normalization, and (c) distributed higher-order optimization methods.

We show excellent scalability of this framework on both cloud as well as HPC clusters, and report on the interplay between bandwidth, network topology and bare metal vs cloud. We deploy this approach to train generative models of sizes hitherto not possible, showing that neural PDE solvers can be viably trained for practical applications. We also demonstrate that distributed higher-order optimization methods are 2-3 $\times$ faster than stochastic gradient-based methods and provide minimal convergence drift with higher batch-size.

Index Terms—Deep generative models; Distributed training; PDEs; Loss functions; Cloud vs HPC; Higher-order optimization

I. INTRODUCTION

Numerical simulation is a critical tool in analysis, optimization, design, and control of complex engineered systems. The status quo has predominantly been describing and modeling of such systems through partial differential equations (PDEs) and their numerical approximations. For increasingly complex engineered applications (aircraft, rockets, autonomous systems, etc.) the availability of fast predictive models becomes critical,

especially if the intent is to use these models for design and/or control (so called model-predictive control, MPC).

Modern deep learning approaches have transformed a host of application areas that involve assimilating large data streams to make useful predictions. There has been increasing interest in leveraging these advances for analysis, optimization, design and control of complex engineered systems ([1], [2], [3], [4], [5], [6]). However, off-the-shelf utilization of deep learning strategies have had limited applicability, primarily due to the following drawbacks:

- *Reliance on abundance of data*: Current ML approaches tend to entirely let data dictate the narrative. As a result, the data requirements for training such systems is very large, which may be a major bottleneck for complex simulations;
- *Lack of generalizability*: They are of narrow scope, i.e., they typically only succeed on the task that they are trained on. Additionally, contextual constraints and domain knowledge known from physical system are left unused.

These key issues have motivated the development of *Scientific Machine Learning* (SciML) strategies that seek to bridge modern deep learning concepts with numerical solutions of PDE's. Recent very exciting advances ([7], [8], [9], [10], [11]) have shown the efficacy of deep networks in solving partial differential equations (PDEs). Specifically, methods as described in [10], [11] rely on convolutional neural networks as a *natural* representation of the domain for a PDE. The reliance on data is reduced by explicitly incorporating notions of symmetry, invariance or constraints into the network (either in the loss function, or in the network definition). This also enables better generalizability (due to the satisfaction of the constraints). By training a deep neural network to act as (an arbitrarily accurate) surrogate for a PDE (either a specific instance, or a class of PDEs), significant gains in computational speed have been shown to be possible. This is because the inference

[§]Equal contribution

stage of neural networks is (near) real time, compared to the cost of training. Thus, given a trained network that acts as a PDE solver for arbitrary boundary and initial conditions, the time-to-solve from a practitioner perspective is simply the time for inference. This is consistent with ML standard practices, where the (non-trivial) cost of training is amortized over the large number of inferences required in, say, model predictive control of complex systems.

While this field is evolving very rapidly, a preliminary taxonomy of ‘neural-PDE’ approaches (through the lens of computational science) is as follows: (a) **PDE instance vs PDE family solvers**: Some approaches focus on improving the numerical linear algebra ([10], [12], [13], [14], [15]), and are limited to a single instance of a PDE, while other strategies ([7], [11], [16], [17]) focus on solving a general class of PDEs. Instance solvers have the advantage of excellent performance, but need to be retrained for each problem realization; (b) **point-wise predictions vs full field predictions**: Some approaches focus on making point wise predictions in the domain ([7], [8], [9], [12], [18], [19], [20]), while others ([1], [11], [16]) make full field predictions. Point-wise predictions have the advantage of easier trainability (since the output is usually a single scalar), but full-field predictions naturally account for boundary conditions. A common bottleneck to these ‘neural-PDE’ approaches is that nearly all of them scale poorly for making predictions on high resolution outputs, prohibiting their use in real-world applications. In particular, higher resolution outputs are needed to model multi-scale phenomena [21], [22]. This serves as the motivation for the work presented here, and we illustrate our developments by training **DiffNet**, a *data-free* conditional generative model, to solve a parametric family of PDEs. DiffNet belongs to the full-field predictions and PDE family solver classification in the taxonomy introduced above. As such, it serves as a canonical example of a complicated neural architecture that predicts full field outputs for a space of initial/boundary conditions defining a PDE class. We specifically focus on training DiffNet to solve the *inviscid Burgers’ equation*, which is a fundamental non-linear PDE with wide applicability in fluid mechanics, gas dynamics and acoustics (i.e. conservation laws with shock formation):

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = 0 \quad (1)$$

We seek to solve this PDE for a one-parameter family of initial conditions defined as

$$u(x, t = 0) = \frac{1}{2}(1 - \cos(2\pi cx)) \quad (2)$$

where $c \geq 0$ is the parameter, and the domain of interest is the unit square, $(x, t) \in [0, 1] \times [0, 0.2]$. Conventional numerical strategies for solving this PDE require some stabilization to gracefully resolve the formation of shocks, and can be computationally expensive for resolved simulations. Fig 1 shows a representative solution (generated via space-time finite element solution) for a 1024×1024 mesh. This took about 400 seconds on 1 SKX node on TACC Stampede2, and serves as our comparative baseline for performance.

DiffNet is a *convolutional generative* neural network that takes in instances of parameterized boundary conditions as input and outputs a full field. In order to train DiffNets, we leverage the form of the PDE and minimize the sum of two losses: PDE residual error, and reconstruction error of the initial and boundary conditions. This approach has two major advantages: (1) we only need to *train a single neural network* for the entire parametric family of initial/boundary conditions and/or coefficients, thus allowing fast inference for users; (2) *being data-free*, we do not need any prior solutions of the PDE. However, our prior experience with DiffNets [23] revealed that training DiffNets for larger domain sizes ($> 512 \times 512$) is often impossible on standard GPUs (even on state-of-the-art NVIDIA Tesla V100’s). Stable training for such generative models also requires large batch sizes which leads to increasingly larger GPU memory requirements.

Our primary contribution is a generalized approach to train such large neural network architectures (that are data free) which can serve as (near) real-time neuralPDE solvers. Our main contributions include (a) a *software framework (called DeepFusion) for data parallel distributed deep learning*, (b) a *hybrid distributed programming approach using OpenMP + MPI for efficient inter/intra node communication*, (c) *leveraging Intel MKLDNN for very fast forward and back propagation*, (d) *synchronized batch normalization*, (e) *loss integrity independent of number of processes*, (f) *support for Hessian based optimization methods*, (g) *illustrating this framework to train DiffNet models for 1024×1024 domain sizes, which was hitherto not possible on GPUs*, and (h) *providing results that show $40\times$ speed-up in time to solve a PDE using DiffNets compared to conventional PDE solvers, considering only inference-time*.

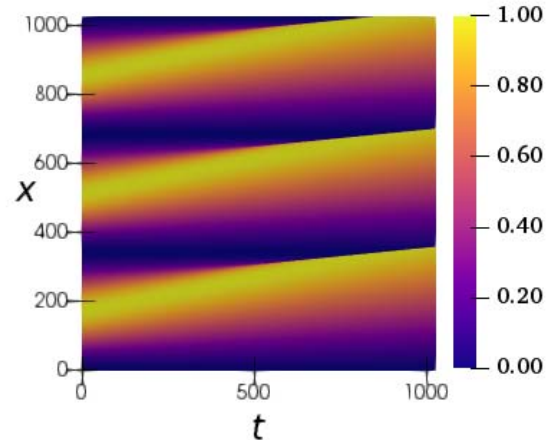


Fig. 1: A solution to the inviscid Burgers’ equation showing shock formation, solved through finite element method formulated in full space-time domain. Physically $(x, t) \in [0, 1] \times [0, 0.2]$ but all contour plots are rendered in a scaled square grid

II. MATHEMATICAL PRELIMINARIES

Using the notation in Hsieh *et. al.* [10], we consider a PDE defined as

$$\mathcal{A}_\nu(\mathbf{u}) = f, \quad \mathcal{B}(\mathbf{u}) = \mathbf{b} \quad (3)$$

where $\mathbf{u}$ is the solution to the PDE over the domain $\Omega \in \mathbb{R}^s$, $\mathcal{A}_\nu$ is the non-linear functional form of the PDE defined by its coefficients ν , and f is a forcing function. Here, $\mathcal{B}(\cdot)$ refers to the boundary conditions for the PDE. Without loss of generality, we assume that Ω is the unit square.

A. DiffNets

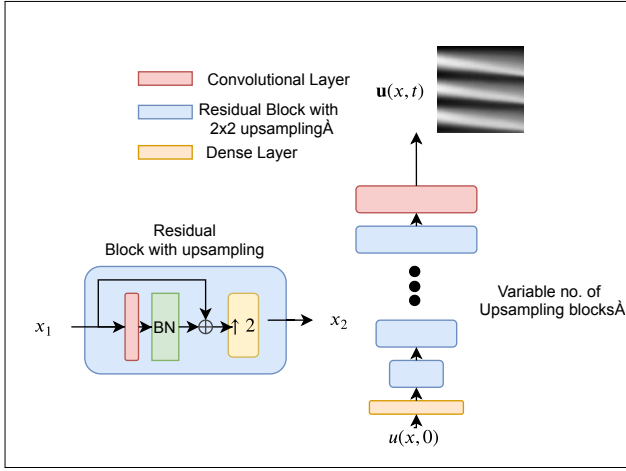


Fig. 2: An exemplar architecture of DiffNets. A specific initial condition, $\mathbf{u}(x,0)$ is given as input to the generative model, which then generates the solution for the specified initial value problem. The primary building blocks for the network include residual blocks with upsampling operations. The number of upsampling blocks in the network depends on the resolution of the domain.

For numerically solving the PDE, the standard approach is to discretize Ω into $\mathbf{S} \in \mathbb{D}^s$ where $\mathbb{D}$ is a discrete subspace of $\mathbb{R}^s$. Subsequently, the field $\mathbf{u}$ can be discretized into a finite dimensional vector, $\bar{\mathbf{u}}$. This is accomplished by using a basis of piecewise-constant functions defined over each discrete element. One can similarly discretize the boundary conditions. Given a guess solution, $\bar{\mathbf{u}}$, standard computing approaches (Finite Difference or Finite Element) linearize the non-linear PDE about this guess solution, defining the PDE Jacobian, $\mathbf{A}_{\bar{\mathbf{u}}}$ and iteratively solve

$$\mathbf{A}_{\bar{\mathbf{u}}}(\delta\bar{\mathbf{u}}) = \text{res}(\bar{\mathbf{u}}); \quad (4)$$

$$\bar{\mathbf{u}} \leftarrow \bar{\mathbf{u}} + \delta\bar{\mathbf{u}} \quad (5)$$

where $\text{res}(\bar{\mathbf{u}})$ is the residual of the current guess w.r.t. the PDE. The key computational cost lies in the repeated solution to the linear equation, Eq. 4, while computing the residual is computationally trivial. Most modern 'NeuralPDE' approaches exploit this computational asymmetry – checking to see if

a guess $\bar{\mathbf{u}}$ is in fact a solution is far more computationally cheaper than actually solving the PDE for the solution.

The DiffNet approach is built on this concept, and is a completely data-free strategy. The solution space is modeled using a generative neural network. The network consists of a generator $G_\theta : \mathbb{R}^k \rightarrow \mathbb{R}^d$ that takes as input the PDE coefficients ν and the initial/boundary conditions $\mathbf{b}$. The generator is trained to produce a solution to the PDE that satisfies these initial/boundary conditions and coefficients. Such an approach can also model the stochastic PDE scenario where $\mathbf{b}$ and ν are stochastic.

For $G_\theta(\cdot)$ to correctly represent the solution, the generator outputs must satisfy: (1) $G_\theta(\cdot)$ must satisfy the PDE, and (2) $G_\theta(\cdot)$ must respect the provided initial/boundary conditions. The training loss can therefore be written in terms of two components:

$$L = L_p + \lambda L_b, \quad (6)$$

$$\text{where } L_p(\theta) = \mathbb{E}_{\mathbf{b}, \nu} [\|\mathcal{A}_\nu(G_\theta(\mathbf{b}, \nu)) - f\|_2^2], \quad (7)$$

$$L_b(\theta) = \mathbb{E}_{\mathbf{b}} \|\mathcal{B}(G_\theta(\mathbf{b}, \nu)) - \mathbf{b}\|_2^2. \quad (8)$$

The first term, L_p , minimizes the residual of the PDE while the second term, L_b , minimizes the deviation from the given initial/boundary conditions. As stated in the introduction, this is the overarching strategy for a variety of neural-PDE solvers (e.g., PINN [8] and other works such as [16]). The distinction of our approach lies in our choice (a) of predicting the full field, $\mathbf{u}(\mathbf{x})$, rather than a single point in the domain. This allows natural enforcement of boundary and initial conditions; and (b) of using a generative model in contrast to other recent approaches. Generative models naturally account for uncertainty, and the network can be extended to produce higher resolution outputs in a straight forward way.

For training, we sample from the space of possible boundary conditions and coefficients, $\{\mathbf{b}_i, \nu_i\}, i = \{1, 2, \dots, k\}$ and minimize the summed loss with respect to θ using stochastic gradient descent. Sampling from a distribution of $\mathbf{b}$ and ν allows the generator to learn the solutions for the family of PDEs parameterized over $(\mathbf{b}, \nu)$. We emphasize that the method is data-free as the training only utilizes the PDE inputs and does not need any PDE solutions for training.

Implementing the forward model. The derivatives of $L_p(\theta)$ with respect to θ require calculating $\frac{\partial \mathcal{A}_\nu}{\partial \theta}$. This is generally non-trivial and to make this tractable we borrow ideas from finite difference methods. We approximate the k^{th} order derivative operator, $\nabla_{(\mathbf{x}, t)}^k$ with convolutional operators defined using finite-difference kernels. In practice, we use 3×3 Sobel kernels [24] for first order derivatives and Laplacian kernels [25] for second order derivatives. This is identical to the approach adopted by Zhu *et. al.* [16]; however, their setup is somewhat restrictive since they use Encoder-Decoder (ED) networks to construct solutions for a given specific PDE.

In the case of time-dependent PDEs, the generator G_θ must learn to first reproduce the initial condition $\mathbf{u}_0$ at $t = 0$ in order to successfully generate the rest of the solution. An incorrect choice of the Lagrangian coefficient, λ , leads to failure either

by the model learning to generate the trivial solution (0) or failing to converge. Additionally, the derivative operators in x and t need to be scaled appropriately in order to satisfy the Courant-Friedrichs-Lewy condition for stability.

We show an exemplar architecture for a DiffNet in Fig. 2. Note that we rely on additional residual upsampling blocks for finer resolution, so as to keep the parameter count low. The advantage of training a conditional generative model is that we only need to train a *single* model for a distribution of parameters characterizing the system. Our approach allows for interpolating and (possibly) extrapolating over unseen boundary conditions and coefficients to generate solutions. While our approach uses convolutional layers to reduce the number of parameters, standard GPU based training still restricts us to solving PDEs for limited domain sizes. However, scaling our method to large scale distributed training allows us to bypass this specific disadvantage. In the following section, we discuss our approach for distributed training of DiffNets.

III. ALGORITHMIC DEVELOPMENTS

GPUs remain the overwhelmingly popular compute platform for training these models. GPU memory utilization during training is driven by three factors: 1) number of model parameters in the network, 2) mini-batch size, and 3) size of intermediate tensors created during loss and gradient computations. A known limitation of GPUs is their relatively small available memory: for example, a state-of-the-art NVIDIA Tesla V100 GPU has only 32GB memory. Peak memory utilization to train a DiffNet for domain size 512×512 and mini-batch size 64 is ~ 64 GB, which is twice the available GPU memory. Due to these memory limitations, training on GPUs is done using mini-batches as small as 16, which in turn results in slow convergence and prohibitive wall-clock times. On Table I, we show maximum batch size and GPU memory utilization for different domain sizes that we were able to train on a NVIDIA Tesla RTX with 24GB memory. *DiffNet training on domain sizes $> 512 \times 512$ is not feasible on currently available GPUs including Tesla V100.*

Domain Size	Batch Size	GPU Memory (GB)	Time/Epoch (s)
128×128	16	0.7	105
256×256	16	2.1	340
512×512	16	16.4	1401

TABLE I: GPU memory utilization and time per epoch for 4096 samples for different domain and batch sizes. On Titan RTX with 24GB memory, Diffnet training on domain sizes $> 512 \times 512$ with batch size 16 is not feasible.

NVIDIA AI Servers like DGX-2 can accommodate bigger batch sizes by distributing the batches across multiple GPUs in a single unit with more cumulative GPU memory (256GB with 8 GPUs and 32GB/GPU). However, they come with an expensive price tag of $\sim \$0.5$ M and are not affordable for the general practitioner. In spite of this price tag, the maximum available memory is **still the same** as the cumulative memory available on 1 or 2 nodes of a CPU cluster.

In order to overcome those memory limitations, our *DeepFusion* framework is based on *data parallel* distributed training on multi-node CPU clusters with 5-10x more memory-per-node than a single GPU, and multiple cores-per-node connected via high-end interconnects with low latency and high bandwidth, which can match or exceed the performance of single GPU. In addition to *data parallelism*, extension to *model parallelism* can further push the envelope on accessible network sizes.

The rest of this section is organized as follows: in section III-A we provide details on data parallel training; in section III-B we discuss the need for synchronized batch normalization; the OpenMP and MPI based hybrid distribution model is explained in section III-C; time complexities for computation and communication are discussed in section III-D, and a comparison to open-source software is made in section III-E. In Table II, we summarize the notations used in this section.

N_s	Total number of samples
b_s	Number of samples in a mini-batch
N_s^{loc}	Local number of samples
b_s^{loc}	Local mini-batch size
N_b	Number of mini-batches
N_w	Number of weights in the model
p	Number of MPI tasks in <i>comm</i>
N_t	Number of threads per MPI task
F	Forward propagation complexity
B	Backward propagation complexity
L	Loss function
θ	Model parameters
G_θ	Generator model

TABLE II: Notations used in this section

A. Data Parallel Distributed Deep Learning

We use the *data parallel* strategy, where multiple replicas of a model are simultaneously trained to optimize a single objective function [26]. In this approach, the training mini-batches are equally split among the available workers, as shown in Figure 3. Each of these workers asynchronously perform forward and back-propagation of their local mini-batch through the neural network. After each mini-batch, the locally computed gradients are averaged among workers via an *MPI_Allreduce* operation, and that average is used by each local optimizer to update the layer parameters. The loss function is also computed locally, and the objective value is averaged among workers.

It is important that the samples (training examples) are split among workers in such a way that the same exact problem gets solved no matter the number of processes (MPI ranks). For that purpose, we adjust the total number of samples N_s and the batch size b_s to make them divisible by the number of workers p , such that the *local* sample count and batch size are given by

$$N_s^{loc} = \lceil \frac{N_s}{p} \rceil, \quad (9)$$

$$b_s^{loc} = \lfloor \frac{b_s}{p} \rfloor. \quad (10)$$

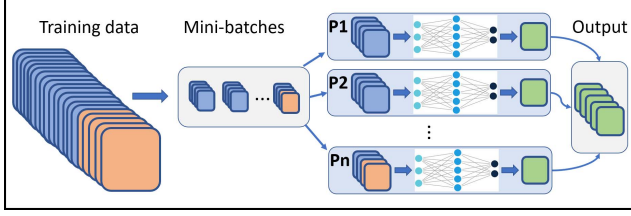


Fig. 3: Data Parallel Deep-Learning : multiple replicas of the model are asynchronously trained by workers, each worker using a subset of the global mini-batch.

Furthermore, as shown in Figure 4, each worker draws their local mini-batch sequentially from the global sample pool, in such a way that their union (*global* mini-batch) would equal the one used in a single-processor run with the same values of N_s and b_s . This guarantees exactly identical results (modulo rounding errors due to gradient reduction) for any number of workers used. Finally, one can easily show that, when N_s is not divisible by b_s , the remainder $N_s \bmod b_s$ will still be divisible by p , which enforces optimal load balancing by guaranteeing that the local mini-batches processed by each worker at any given time have identical sizes.

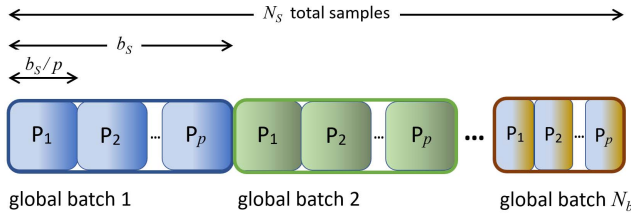


Fig. 4: Data splitting across workers in a parallel run: mini-batches are always guaranteed to have the same size across workers at any given time, promoting optimal load balance.

Algorithm 1 summarizes our per-epoch strategy for distributed training using the data parallel paradigm.

Algorithm 1 Data parallel distributed deep learning (1 epoch)

Require: Generate pN_s^{loc} samples, split in mini-batches of size b_s^{loc}

- 1: **for** $mb = 1$ **to** N_b **do**
- 2: $\mathbf{x}_i \leftarrow \{b_s^{loc} \text{ samples}\}$ ▷ Worker i local mini-batch
- 3: $\ell_i(\theta) \leftarrow L(\mathbf{x}_i, G_\theta(\mathbf{b}, \nu))$ ▷ Forward pass and local loss
- 4: $\mathbf{g}_i(\theta) \leftarrow \nabla \ell_i(\theta)$ ▷ Back-prop and local gradient
- 5: $\ell(\theta) \leftarrow \frac{1}{p} \sum \ell_i(\theta)$ ▷ Average loss using MPI_Allreduce
- 6: $\mathbf{g}(\theta) \leftarrow \frac{1}{p} \sum \mathbf{g}_i(\theta)$ ▷ Average grad using MPI_Allreduce
- 7: $\Delta\theta \leftarrow u(\mathbf{g}, \theta, t)$ ▷ Run local optimizer
- 8: $\theta \leftarrow \theta + \Delta\theta$ ▷ Update network parameters
- 9: **end for**

B. Synchronized Batch-normalization

Batch Normalization (BN) is a procedure that dramatically improves the convergence of neural networks by re-scaling and re-centering data using running statistics, namely mean and variance, accumulated from each mini-batch in the course of an epoch [27]. This creates a dependency on the local mini-batch size, breaking the paradigm of problem independence on

the number of workers discussed in section III-A. To remove this dependency, we developed a scheme to synchronize the mean and variance statistics at all BN layers by performing an *MPI_Allreduce* operation after each epoch. This is especially important when the local mini-batch size on each processor is small, which would result in poor statistics and have a negative impact on validation accuracy. Our BN synchronization scheme is explained in Algorithm 2. Although similar to the idea proposed in [28], our implementation is optimized to only communicate batch statistics before inference, which reduces communication costs without any qualitative change in results.

Algorithm 2 Batch-normalization synchronization algorithm

- 1: **for** epoch **do**
- 2: **for** BN layer in network **do**
- 3: **if** in evaluation **then**
- 4: $\mu_B \leftarrow \frac{1}{p} \sum_i \mu_B^{(i)}$ ▷ Allreduce BN means
- 5: $\sigma_B^2 \leftarrow \frac{1}{p} \sum_i \sigma_B^{2(i)}$ ▷ Allreduce BN variances
- 6: **end if**
- 7: **end for**
- 8: **end for**

C. Hybrid Distribution Model

Our parallel distribution scheme is based on the so called *hybrid MPI-OpenMP* programming paradigm, in which communication between processes is done via MPI, while each process can spawn its own OpenMP threads that run inside a single shared-memory processor (SMP) node, as illustrated in Figure 5. Furthermore, since the OpenMP threads only communicate with other threads within the same SMP node, and MPI routines are only invoked outside of OpenMP parallel regions, our distribution scheme can be said to model the *process-to-process* hybrid paradigm. In particular, our application spawns p processes via the usual *mpirun* utility, which can land on up to p SMP nodes. The number of processes per node depends on the specific specs of the host machines and on details of the experiment. A few underlying libraries used by our application (e.g., libtorch and mkldnn) spawn up to N_t local threads of their own, where N_t can be controlled via the *OMP_NUM_THREADS* environment variable.

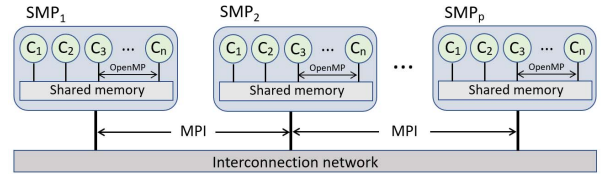


Fig. 5: Process-to-process hybrid distribution paradigm: processes communicate via MPI, and spawn local threads of their own that communicate via OpenMP inside an SMP node. MPI routines are only invoked outside of OpenMP parallel regions.

D. Complexities

In this sub-section, we will discuss computation and communication complexities of our approach. As shown in Figure

3, the entire data set with N_s samples is split into N_b mini-batches with b_s samples per mini-batch. Each MPI task computes loss and gradients for the local mini-batch of size b_s^{loc} . Loss and gradient computations have a complexity of

$$O(F(N_w, b_s^{loc}) + B(N_w, b_s^{loc})). \quad (11)$$

Forward (F) and backward (B) propagation complexities are non-linear functions of N_w, b_s^{loc} , number of cores allocated per MPI task and the network architecture. We use libtorch C++ APIs to execute forward and backward propagation in a single MPI task, which internally uses MKLDNN [29] for optimal performance on Intel CPUs. Local gradients are averaged using *MPI_Allreduce*, which has a communication complexity of $O(N_w + \log(p))$. Since $N_w \gg p$, we expect the communication time to remain almost constant, independent of p and of the underlying algorithm used by the OpenMPI implementation. Network weights on each process are updated using this averaged gradient via a stochastic gradient descent (SGD) method. Synchronization of statistics for all batch normalization layers after each epoch has a complexity of $O(N_w)$. The overall complexity per epoch is, therefore,

$$O(N_b (F(N_w, b_s^{loc}) + B(N_w, b_s^{loc}) + N_w)). \quad (12)$$

E. Comparison to Open Source Software

Tensorflow and **Pytorch** are the most popular open source libraries for deep learning, and both provide support for distributed deep learning. Out of the box, they handle all-reduce operations on gradients computed on a local mini-batch on each device. **Horovod** is an open source library [30] from Uber that enables data parallelism and gradient averaging. However, optimal load balancing, guarantees of loss independence on processor count, p (Figure 8), synchronized batch normalization and hybrid distributed implementation discussed in the previous sections are not provided out of the box by these open source libraries.

DeepFusion democratizes the ability of data scientists to train models that are too big for GPUs with desired system performance and convergence rates without any distributed and high performance computing experience.

Out of the box functionality	TF	Pytorch	Horovod	DeepFusion
All reduce on Gradient	Yes	Yes	Yes	Yes
Loss Integrity independent of p	No	No	No	Yes
Synchronized Batch Normalization	No	No	No	Yes

TABLE III: Qualitative comparison of DeepFusion functionality with Open Source Software

IV. RESULTS AND DISCUSSIONS

One of the key outcome of our experiments was to demonstrate a practical approach to train DiffNets on domain sizes $> 512 \times 512$ that are too big for GPUs. We tested our framework on both TACC **Stampede2** HPC clusters with

bare-metal access, as well as **Microsoft Azure** and **Amazon Web Services** (AWS) HPC clusters built using on-demand virtual machines. We target these computational resources as representative of what is easily accessible for the general data science practitioner unlike DGX-2 that requires significant investment. We report results obtained from AWS, Microsoft Azure and Stampede2. On Table IV, we provide all relevant specifications for Azure and Stampede2 used in our experiments. Care was taken to select configurations on AWS and Azure to reasonably match the CPU as well as interconnect speeds of Stampede2. This allows rational assessment of performance of DeepFusion across nearly similar platforms. We present wall-clock time comparisons between AWS, Azure and Stampede2 in section IV-A to determine the cluster to use for our large domain runs. In section IV-B, we conduct strong scaling experiments for 128×128 and 256×256 domain sizes for 1 to 128 nodes (48 to 6144 cores) on Stampede2 to demonstrate scalability of our software. Finally, in section IV-C, we present results for training a DiffNet model for 512×512 and 1024×1024 domain sizes for Burgers' inviscid equation (Eq. 1) for parameter distributions characterizing the initial conditions (Eq. 2), which are currently not possible to train.

Specification	AWS	Azure	Stampede2
Type	Virtual Machine	Virtual Machine	Bare-Metal
CPU	Intel Xeon Platinum 8000	Intel Xeon Platinum 8168	Intel Xeon Platinum 8160
CPU cores	72	44	48
Memory (GB)	192	352	192
Interconnect	Elastic Fabric Adapter	EDR Infiniband	Intel Omni-Path
Bandwidth	100 Gb/sec	100 Gb/sec	100 Gb/sec
Topology	AWS Proprietary	Fat tree	Fat tree

TABLE IV: Functional specifications of AWS, Microsoft Azure and Stampede2 infrastructure used in our experiments.

A. Conventional HPC vs. Cloud Based HPC

In this section, we compare wall-clock times between AWS, Microsoft Azure, and Stampede2 in order to determine the optimal HPC cluster configuration to train *DiffNet* with very large (1024×1024) resolutions. On Table V, we show per-epoch wall-clock times (in seconds) to train DiffNet with 64×64 resolution using 1, 2 and 4 nodes. The total number of samples used for this experiment was $N_s = 4096$ and the global batch size was $b_s = 1024$; the number of processes per node was fixed at 4, with each process spawning 8 local threads. Single node performance on bare-metal Stampede2 is $\sim 2\times$ faster than on Azure and AWS VM. On the same table, we also compare per-epoch wall-clock times (in seconds) for different resolutions on 4 compute nodes (with 4 processes per node, 8 threads per process). Even though the infrastructure specifications of Azure, AWS and Stampede2 are almost identical, slowness on Azure and AWS can be attributed to the overheads associated with virtual machines.

Domain Size	Nodes	AWS	Azure	Stampede2
64x64	1	131.0	113.1	67.2
64x64	2	65.2	54.9	34.9
64x64	4	32.4	28.6	19.4
128x128	4	138.4	126.2	68.5
256x256	4	650.5	597.6	279.8

TABLE V: Comparison of per-epoch wall-clock times (in seconds) between AWS, Azure and Stampede2 for varying resolutions and different number of nodes (see Table IV for cluster specs). For all three clusters, 4 processes were used per node (spawning 8 threads each).

B. Scaling Experiments

In Figure 6, we report strong scaling results to train *DiffNet* for 128×128 and 256×256 resolutions, using from 4 to 128 nodes on Stampede2. In this experiment, we used 8 MPI processes per node and each process spawned 12 threads, to a total of 96 threads per node. This matches the full capacity of the Stampede2 Skylake nodes, which have 48 physical hyperthread-enabled CPU cores, resulting in 96 hardware threads per node. In Table VI, we compare per-epoch wall clock time between a single GPU (Titan RTX as well as Tesla V100) with the wall clock time using 128 Stampede2 nodes. We show this (potentially unfair) comparison to illustrate the advantage of scale-up on CPUs using a distributed training approach.

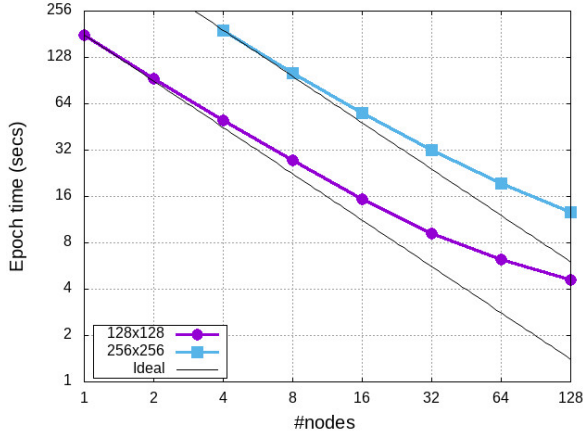


Fig. 6: Strong scaling results for training *DiffNet* on Stampede2: per-epoch times (in seconds) versus number of nodes for 128×128 and 256×256 resolutions, using 8 processes per node and 12 threads per process.

As discussed in section III-D, our computation time complexity scales with p and communication complexity is independent of p . In Figure 7, we show the computation and communication wall-clock times for different p . We observed that our communication times increase only slightly with p , as expected. Note that the communication times are significantly ($100\times$) smaller than compute times.

As discussed in section III-A, the training samples are split

Output resolution	1 Titan RTX (seconds)	1 Tesla V100 (seconds)	128 Stampede2 nodes (seconds)
128×128	105	130	4.6
256×256	340	494	12.7
512×512	1401	1961	25.3
1024×1024	N/A	N/A	89.5

TABLE VI: Comparison of per-epoch wall-clock time between Titan RTX, Tesla V100 and DeepFusion on 128 Stampede2 nodes to train Diffnet (of different resolutions) with 4096 samples. Training on large CPU clusters using DeepFusion is $20 - 60\times$ faster than training on both GPU's. The 30-40% change between the Titan RTX vs Tesla V100 is attributable to the 30% difference in clockspeed between them.

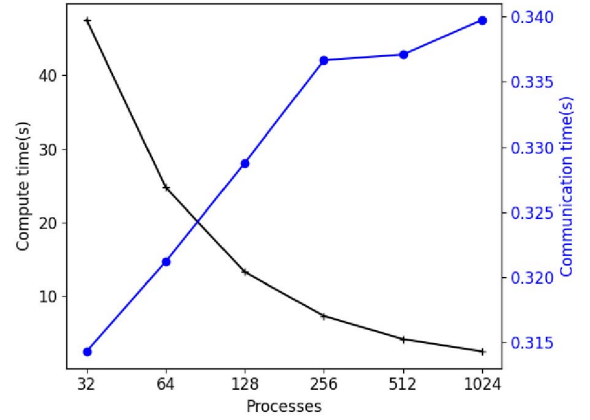


Fig. 7: Computation and communication wall clock times for different p corresponding to strong scaling results for domain size 256×256 in Figure 6. Computation time decreases linearly with p , while communication time increases only slightly with p .

in such a way as to guarantee loss decay integrity, i.e., the same exact problem is solved independently of the number of MPI processes. In Figure 8, we show the loss vs. epoch for different values of p . The small deviation in loss values for different p is due to rounding errors in *MPI_Allreduce* operations for computing gradient averages.

C. High Resolution DiffNet

In this sub-section, we illustrate the ability of the framework to train *DiffNet* models with very high resolution outputs (sizes $\geq 512 \times 512$). This typically requires ≥ 2000 epochs until convergence (for different ranges of the initial condition frequency c) using a first-order optimizer like SGD.

Example 1: In the first example, we train a *DiffNet* to produce outputs of resolution 1024×1024 . We emphasize that generative models of this size have hitherto fore not been trained (to the best of our knowledge). We train the *DiffNet* to be predictive in a range of the parameter $c \in [3, 6]$. We remind the reader that c represents a one-parameter family of initial conditions to the inviscid Burgers equation. The training

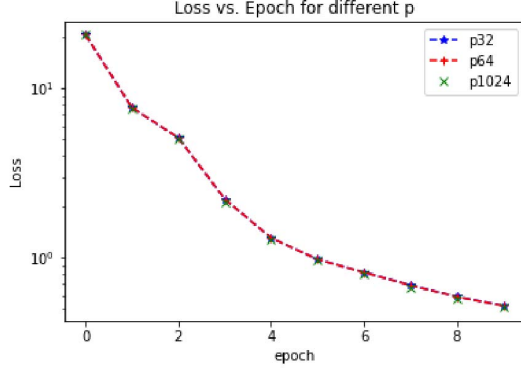


Fig. 8: Comparison of loss decay vs. epoch for a subset of p used in strong scaling experiments in Figure 6. Loss decay is preserved (independent of p).

sample set consists of 256 points from $c \in [3, 6]$. We set the mini-batch size to 64. This model was trained on 8 nodes of Stampede2 (with 8 processes per node), taking 2200 epochs till convergence (see Fig. 10). The runtime for this training was 32 hours.

After training, we performed inference using the DiffNet for a set of initial conditions from the one-parameter family. Results for $c = 3$ are shown in Fig. 9, where we compare the DiffNet inference result with the solution from an optimized FEM solver (based on the Petsc library) of the inviscid Burgers equation. While the general trend of the solution (as the wave evolves – from left to right in the figures – and forms shocks) is captured well, there is still room for improvement as the more diffused nature of the ML solution indicates. We hypothesize that using higher order Sobel filters (i.e. computing the gradients using higher order stencils) could help in eliminating the diffusive features of the ML solution. We continue to explore these aspects.

Example 2: In the second example, we explore if the DiffNet can be trained to predict solutions for a much larger distribution of the initial conditions. We train a DiffNet to predict solutions at 512×512 resolution, but for initial conditions from $c \in [3, 16]$. At higher values of c (which represent initial conditions exhibiting higher frequencies), we expect the formation of multiple shocks. The traditional numerical solutions for these initial conditions have to be carefully performed. We set the mini-batch size to 64. This model was trained on 8 nodes (64 processors) on Stampede2, taking 4000 epochs till convergence (see Fig. 11). The runtime for training this model was 15 hours. Notice that the loss in this case is significantly larger than the previous example. This is due to two reasons: the reduced resolution ($1024 \rightarrow 512$) and, more importantly, the larger c space.

After training, we again performed inference using the DiffNet for a set of initial conditions from the one-parameter family. Results for $c = 3, 5, 10, 13$ are shown in Fig. 15, where we compare the DiffNet inference result with the solution from

N	c	$\ u_g\ _2$	$\ u_{fd}\ _2$	$\ u_{fe}\ _2$	$\ \delta_{fd}^g\ _2$	$\ \delta_{fd}^{fe}\ _2$
512	3	0.506	0.570	0.570	0.134	0.007
512	5	0.495	0.527	0.527	0.077	0.009
512	10	0.497	0.493	0.493	0.038	0.013
512	13	0.507	0.493	0.493	0.043	0.016
1024	3	0.544	0.570	0.570	0.063	0.004

TABLE VII: Norm of different solutions (denoted u) and their differences (denoted δ). u_g is the solution generated through DeepFusion, u_{fd} is the solution obtained using explicit time marching with finite difference approximation; and u_{fe} is the “space-time” solution obtained through finite element approximation in *both* space and time. The differences between them: $\delta_{fd}^g = u_g - u_{fd}$ and $\delta_{fd}^{fe} = u_{fe} - u_{fd}$. All norms are calculated over the entire spatiotemporal domain

an optimized FEM solver (based on the Petsc library) of the inviscid Burgers equation. As before, the general trend of the solution (as the wave evolves – from left to right in the figures – and forms shocks) is captured well, but there is still room for improvement.

Fig. 12 and Table VII show additional, quantitative comparison between the DiffNet results with a stabilized Finite Element solution (at 512×512 resolution) against a very high resolution (2048×2048) finite difference solution. Fig. 12 plots the solution at one time point ($t = 0.2$), and suggests that stabilized finite element space-time approach is still unable to capture the shocks, while the DiffNet is able to accurately capture the shock without any dispersive effects. This is particularly promising as the loss function used in the DiffNet is the simplest one possible, with significant room for improvement. Table VII shows the L2 error norm (defined as the sum of squared error of the solutions in space-time) of the DiffNet and FEM solution against the high resolution FDM solution (which we take as our ‘reference’ solution). The relative error ($\|\delta_{fd}^g\|_2 / \|u_{fd}\|_2$ (for the worst case of $c = 3$) drops from 0.23 for $N = 512$ to 0.11 for $N = 1024$ making the case for the need for networks that produce higher resolution outputs. Interestingly, we find that the DiffNet approach produces more accurate results for initial conditions exhibiting more waves (larger c). This is in contrast to what is observed in traditional approaches (compare the last two columns of Table VII). We find it promising that the DiffNet is able act as a general PDE solver for a wide range of initial conditions. This strongly suggests that, with the proper training infrastructure, it is possible to develop truly general PDE solvers that produce accurate solutions for general classes of PDE’s.

Effect of batch size on solution using first order (SGD) and second order (L-BFGS) optimizers: The data parallelism afforded by DeepFusion potentially allows one to use larger batch sizes. However, it is well known that increasing batch sizes can decrease the convergence of Stochastic Gradient Descent (SGD). We explore this effect of batch size on the training performance. Fig. 13(top) plots evolution of the loss function with training epochs for increasing batch size (BS) for a 64×64 resolution DiffNet. We clearly see some degradation in convergence rate as the batch size is increased

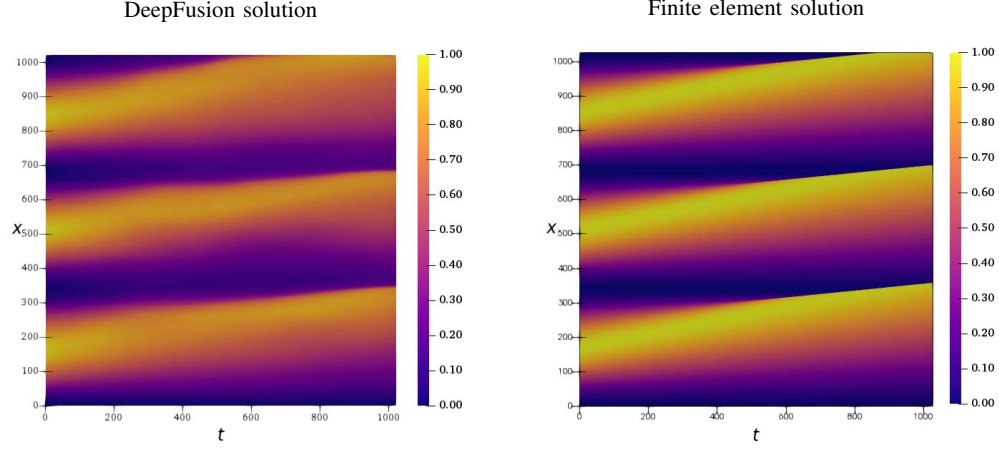


Fig. 9: Inviscid Burgers' equation solved on 1024×1024 pixels (left) or elements (right)

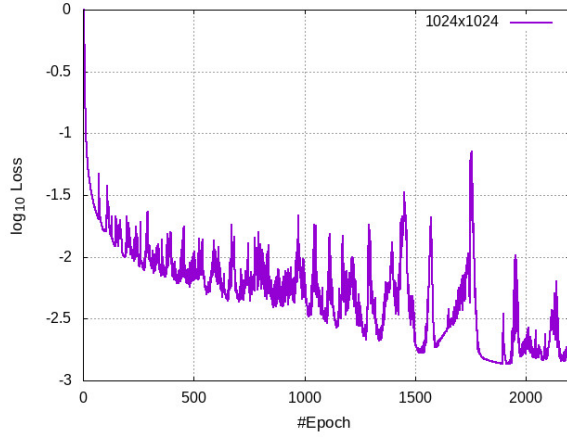


Fig. 10: Loss vs. epoch for training DiffNet with domain size 1024×1024 on Stampede2 using 256 sample points and batch size 64.

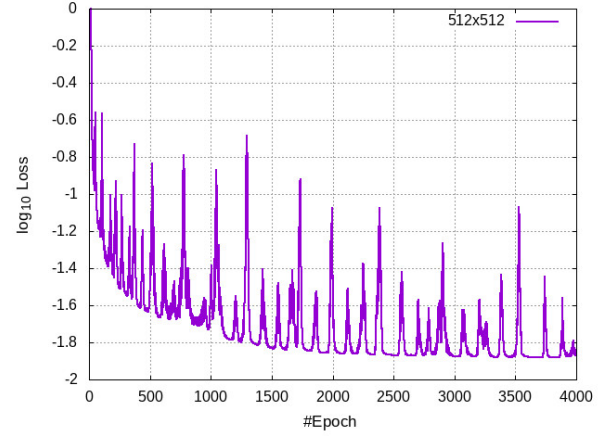


Fig. 11: Loss vs. epoch for training DiffNet with domain size 512×512 on Stampede2 using 256 sample points and batch size 64.

to 64. We next trained the same network using a second order optimizer (L-BFGS) implemented in the DeepFusion framework. Fig 13(bottom) shows negligible impact of increasing batch size on convergence. As expected, the second order method converges in fewer epochs, with similar reduction in loss happening within 10 training epochs as compared to 150 epochs for SGD. L-BFGS optimizers require larger memory (to evaluate the Hessian) which a distributed approach (like DeepFusion or LBANN) can gracefully accommodate.

The increased computational overhead results in increased time per epoch, and this is plotted in Fig. 14. While L-BFGS takes an order of magnitude less number of epochs to converge, each epoch is more expensive due to memory and compute requirements from evaluating the Hessian. It is informative also to look at the results in terms of computational time to reach a certain convergence threshold, with L-BFGS schemes about $3\times$ faster than the SGD scheme. This reduction in computational time to reach a desired loss threshold can be enhanced via parallelization. DeepFusion allows

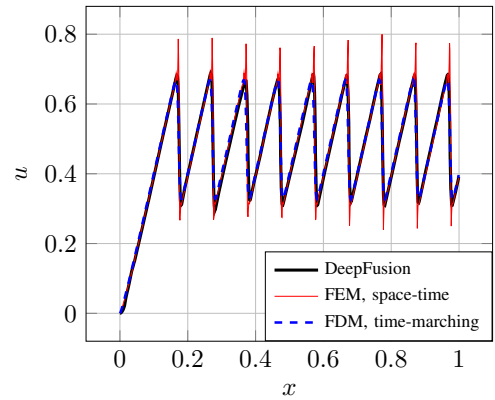
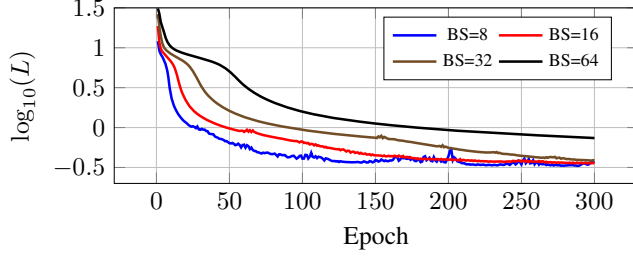
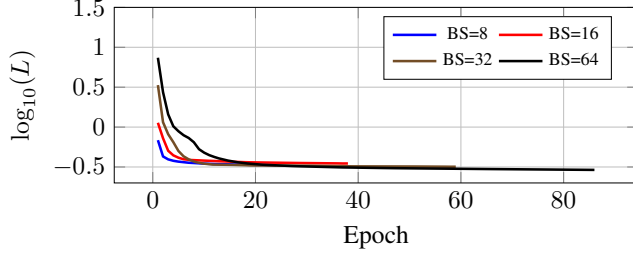


Fig. 12: An example of the solution profile u vs. x at the final time ($t = 0.2$), compared for different methods. Resolution = 512×512 and $c = 10$

parallelization of L-BFGS based training across multiple CPU



(a) SGD



(b) L-BFGS

Fig. 13: Loss vs. epoch for 64×64 domain DiffNet with varying batch-size (BS). Second order optimizers (L-BFGS) provide minimal convergence drift with higher batch-size. All training done on 1 node of Nova

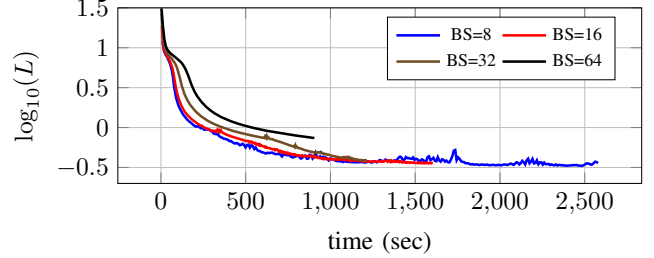
p	64×64	128×128
2	63.6	208.4
4	42.5	113.4
8	32.0	67.2
16	23.6	46.2

TABLE VIII: Time (in minutes) taken to complete 100 epochs of the L-BFGS method for the 64×64 and 128×128 domain DiffNet on different number of processors

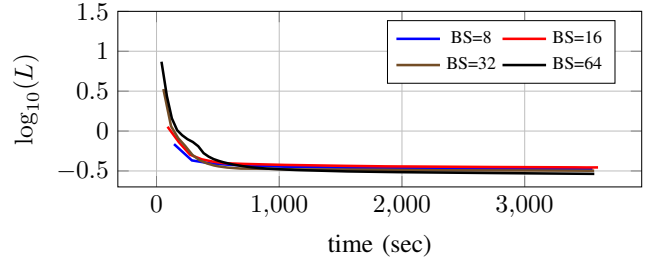
nodes, and this training across multiple nodes proportionally decreases the time-to-train, as shown in Table VIII

These results are very promising as they allow using second order methods – which are less sensitive to large batch sizes – and data parallelization to reduce time-to-solve, which in turn, will allow us to solve such problems faster compared to SGD based approaches.

Comparison of run-times between DiffNet and FEM based solutions: While the training times for ML models is admittedly long, once trained, the inference step is often very fast. This allows one to amortize the cost of training across multiple users and instances. The availability of a general ‘NeuralPDE’ makes this a viable possibility. We quantify this argument by comparing the time it takes for a trained DiffNet to make a prediction (i.e. inference) with the time it takes for a well optimized FEM solver to perform the same prediction. Both DiffNet inference and FEM solve are performed on one node of Stampede2. This comparison is reported in Table IX, which shows a 40x improvement in prediction time. We emphasize that the inference step is not optimized, suggesting that the 40x improvement we show is a lower bound.



(a) SGD



(b) L-BFGS

Fig. 14: Loss vs. time to solve for 64×64 domain DiffNet with varying batch-size (BS). Second order optimizers (L-BFGS) are substantially slower than SGD. All training done on 1 node of Nova

Domain Size	FEM (seconds)	DeepFusion (seconds)
512×512	23.2	3.6
1024×1024	395.6	9.8

TABLE IX: Comparison of solve time for the finite element solution and the inference time for the DeepFusion solution

V. CONCLUSIONS

We report on a data distributed computing approach for training large neural network architectures, especially in the context of *data-free generative models* that serve as PDE solvers. We have addressed twin challenges of GPU memory limitations and wall-clock times for large-scale training. Key contributions of our framework over other data-parallel approaches include loss integrity independent of the number of processes, synchronized batch normalization, and higher-order optimization methods. We demonstrated excellent scaling on current supercomputers. Our software allows practitioners to train very large models, thus enabling practical applications of such **data-free** ‘neuralPDE’ solvers. We believe that availability of such approaches will democratize the ability to produce (near) real time predictions of complex systems characterized by PDEs. Our future goals include extension of the framework to incorporate model parallelism for increased scaleup, as well as apply second order strategies to train DiffNets for a wide range of PDE’s. Other tasks include exploring (a) more complex loss functions (e.g. variational loss), and (b) more complex PDE’s like Navier-Stokes, where the performance of such methods remain unclear.

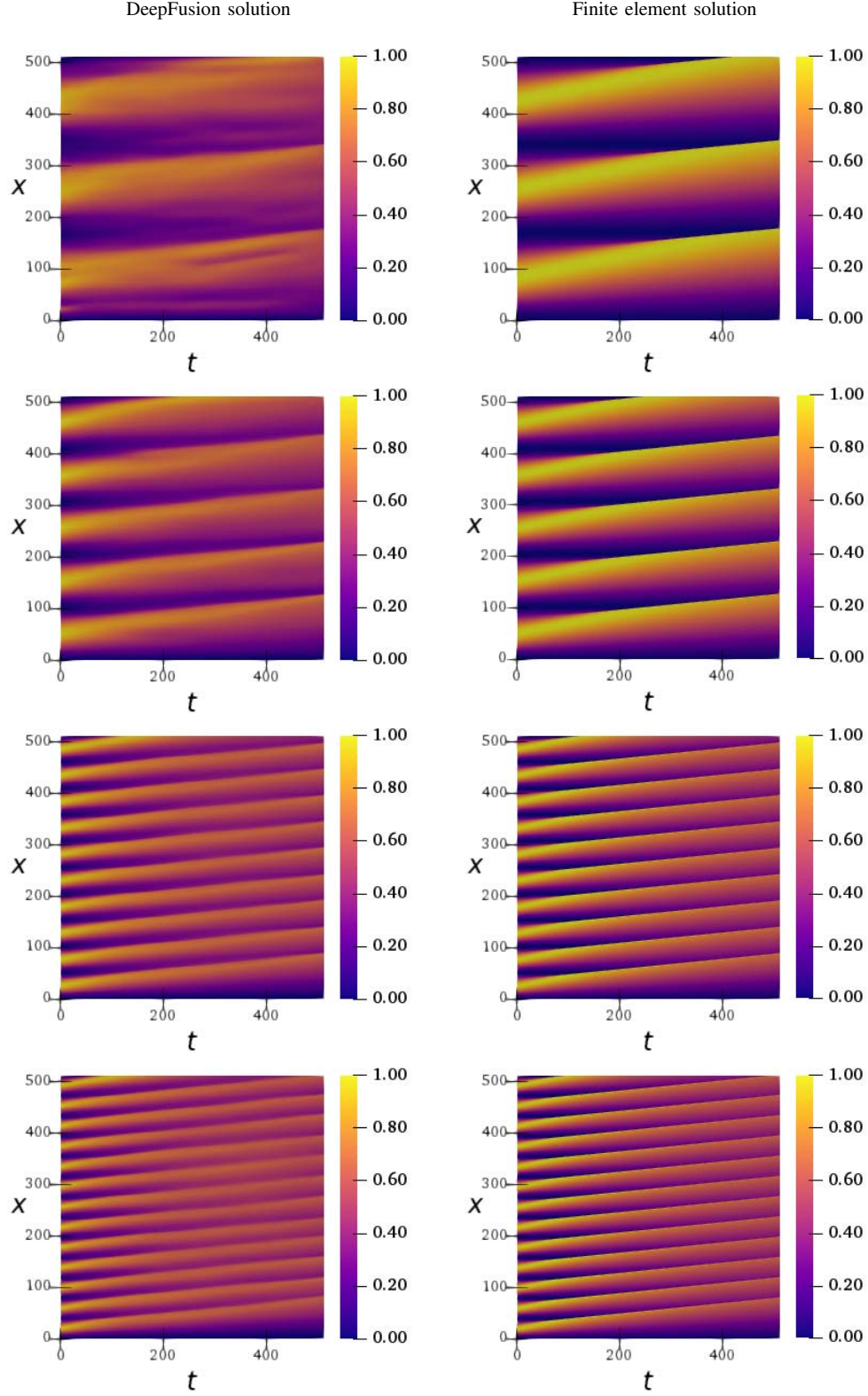


Fig. 15: Contour plots of the solution to the inviscid Burgers' equation: solutions from DiffNet (left) vs solutions from a finite element solver (right). The initial condition is characterized by the wave number c (defined in 2). The first row presents solution for $c = 3$ and subsequently for $c = 5, 10$ and 13 in the latter rows respectively. On the left, the image size is 512×512 pixels, whereas on the right, the discretization is a mesh of 512×512 bilinear quadrilateral elements.

VI. ACKNOWLEDGEMENTS

We acknowledge support from ARPA-E DIFFERENTIATE program (DE-AR0001215), and computing support from XSEDE and Iowa State University. BG and BK also acknowledge partial support from NSF 1935255.

REFERENCES

- [1] M. Paganini, L. de Oliveira, and B. Nachman, “Calogan: Simulating 3d high energy particle showers in multilayer electromagnetic calorimeters with generative adversarial networks,” *Physical Review D*, vol. 97, no. 1, p. 014021, 2018.
- [2] R. King, O. Hennigh, A. Mohan, and M. Chertkov, “From deep to physics-informed learning of turbulence: Diagnostics,” *arXiv preprint arXiv:1810.07785*, 2018.
- [3] C.-W. Chang and N. Dinh, “A study of physics-informed deep learning for system fluid dynamics closures,” in *American Nuclear Society Winter Meeting, Anaheim, CA*, 2016, pp. 1785–1788.
- [4] G. Pun, R. Batra, R. Ramprasad, and Y. Mishin, “Physically-informed artificial neural networks for atomistic modeling of materials,” *arXiv preprint arXiv:1808.01696*, 2018.
- [5] L. de Oliveira, M. Paganini, and B. Nachman, “Learning particle physics by example: location-aware generative adversarial networks for physics synthesis,” *Computing and Software for Big Science*, vol. 1, no. 1, p. 4, 2017.
- [6] B. Sanchez-Lengeling and A. Aspuru-Guzik, “Inverse molecular design using machine learning: Generative models for matter engineering,” *Science*, vol. 361, no. 6400, pp. 360–365, 2018.
- [7] L. Yang, D. Zhang, and G. E. Karniadakis, “Physics-informed generative adversarial networks for stochastic differential equations,” *arXiv preprint arXiv:1811.02033*, 2018.
- [8] M. Raissi, P. Perdikaris, and G. Karniadakis, “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations,” *Journal of Computational Physics*, vol. 378, pp. 686 – 707, 2019. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0021999118307125>
- [9] M. Raissi and G. E. Karniadakis, “Hidden physics models: Machine learning of nonlinear partial differential equations,” *Journal of Computational Physics*, vol. 357, pp. 125–141, 2018.
- [10] J. Hsieh, S. Zhao, S. Eismann, L. Mirabella, and S. Ermon, “Learning neural PDE solvers with convergence guarantees,” in *Proc. Int. Conf. Learning Representations (ICLR)*, 2019.
- [11] A. Joshi, V. Shah, S. Ghosal, B. Pokuri, S. Sarkar, B. Ganapathysubramanian, and C. Hegde, “Generative models for solving nonlinear partial differential equations,” in *Proc. of NeurIPS Workshop on ML for Physics*, 2019.
- [12] G. Pang, L. Lu, and G. E. Karniadakis, “fPINNs: Fractional physics-informed neural networks,” *SIAM Journal on Scientific Computing*, vol. 41, no. 4, pp. A2603–A2626, 2019.
- [13] Z. Long, Y. Lu, X. Ma, and B. Dong, “Pde-net: Learning pdes from data,” in *ICLR*, 2017.
- [14] D. Greenfeld, M. Galun, R. Basri, I. Yavneh, and R. Kimmel, “Learning to optimize multigrid pde solvers,” in *ICML*, 2019.
- [15] A. Katrutsa, T. Daulbaev, and I. Oseledets, “Deep multigrid: learning prolongation and restriction matrices,” *arXiv preprint arXiv:1711.03825*, 2017.
- [16] Y. Zhu, N. Zabarar, P.-S. Koutsourelakis, and P. Perdikaris, “Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data,” *arXiv preprint arXiv:1901.06314*, 2019.
- [17] J. Sirignano and K. Spiliopoulos, “Dgm: A deep learning algorithm for solving partial differential equations,” *Journal of Computational Physics*, vol. 375, pp. 1339–1364, 2018.
- [18] S. Karumuri, R. Tripathy, I. Bilonis, and J. Panchal, “Simulator-free solution of high-dimensional stochastic elliptic partial differential equations using deep neural networks,” *Journal of Computational Physics*, vol. 404, p. 109120, 2020.
- [19] J. Han, A. Jentzen, and E. Weinan, “Solving high-dimensional partial differential equations using deep learning,” *Proceedings of the National Academy of Sciences*, vol. 115, no. 34, pp. 8505–8510, 2018.
- [20] C. Michoski, M. Milosavljevic, T. Oliver, and D. Hatch, “Solving irregular and data-enriched differential equations using deep neural networks,” *arXiv preprint arXiv:1905.04351*, 2019.
- [21] O. Wodo and B. Ganapathysubramanian, “How do evaporating thin films evolve? unravelling phase-separation mechanisms during solvent-based fabrication of polymer blends,” *Applied Physics Letters*, vol. 105, no. 15, p. 153104, 2014.
- [22] B. Ganapathysubramanian and N. Zabarar, “On the control of solidification using magnetic fields and magnetic field gradients,” *International journal of heat and mass transfer*, vol. 48, no. 19-20, pp. 4174–4189, 2005.
- [23] V. Shah, A. Joshi, S. Ghosal, B. Pokuri, S. Sarkar, B. Ganapathysubramanian, and C. Hegde, “Encoding invariances in deep generative models,” *arXiv preprint arXiv:1906.01626*, 2019.
- [24] I. Sobel and G. Feldman, “A 3x3 isotropic gradient operator for image processing,” pp. 271–272, 1968.
- [25] R. C. Gonzalez and R. E. Woods, *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., 2006.
- [26] T. Ben-nun and T. Hoeftler, “Demystifying parallel and distributed deep learning: An in-depth concurrency analysis,” *arXiv preprint arXiv:1802.09941v2*, 2018.
- [27] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” *arXiv preprint arXiv:1502.03167v3*, 2015.
- [28] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, “A simple framework for contrastive learning of visual representations,” *arXiv preprint arXiv:2002.05709v3*, 2020.
- [29] “Intel mkldnn,” <https://01.org/dnnl>, accessed: 2020-04-16.
- [30] A. Sergeev and M. D. Balso, “Horovod: fast and easy distributed deep learning in TensorFlow,” *arXiv preprint arXiv:1802.05799*, 2018.

VII. APPENDIX

A. Batch size effect on convergence

We continue exploring the effect of batch-size on convergence rates of both first order optimizers (SGD) as well as second order optimizers (L-BFGS) both of which are implemented in a data parallel way in DeepFusion. Fig. 16 plots the loss evolution with training epochs for a 128×128 DiffNet model that was trained on 1 node on Nova. These results are consistent with those shown in Fig. 13 for a 64×64 resolution DiffNet model, where L-BFGS optimizer is relatively insensitive to the batch-size ranges chosen. As expected, the second order method converges in fewer epochs, with similar reduction in loss happening within 30 training epochs as compared to 150 epochs for SGD. L-BFGS optimizers require larger memory (to evaluate the Hessian) which a distributed approach (like DeepFusion) can gracefully accommodate. The increased computational overhead results in increased time per epoch, and this is plotted in Fig. 17.

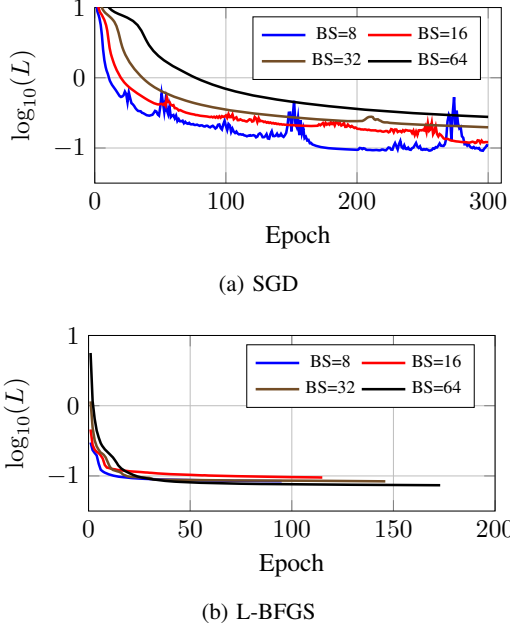


Fig. 16: Loss vs. epoch for 128×128 domain DiffNet with varying batch-size (BS). Second order optimizers (L-BFGS) show faster convergence rate and provide minimal convergence drift with higher batch-size.

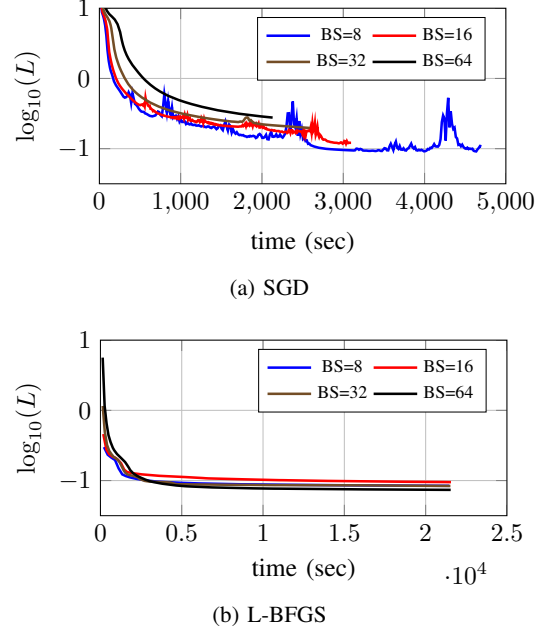


Fig. 17: Loss vs. time for 128×128 domain DiffNet shown in Fig. 16

B. Additional quantitative comparison between DiffNet and conventional PDE solvers

In this subsection, we provide additional results over those shown in the main text to quantitatively compare the DiffNet inferences with conventional PDE solver technology. Figure 18 plots the solution at a particular time instance ($t = 0.2$) where there is formation of shocks. Notice that the DiffNet solution is very close to the fully resolved FDM solution for the large wave-number case ($c = 13$), with increasingly large deviations as the wave-number is decreased.

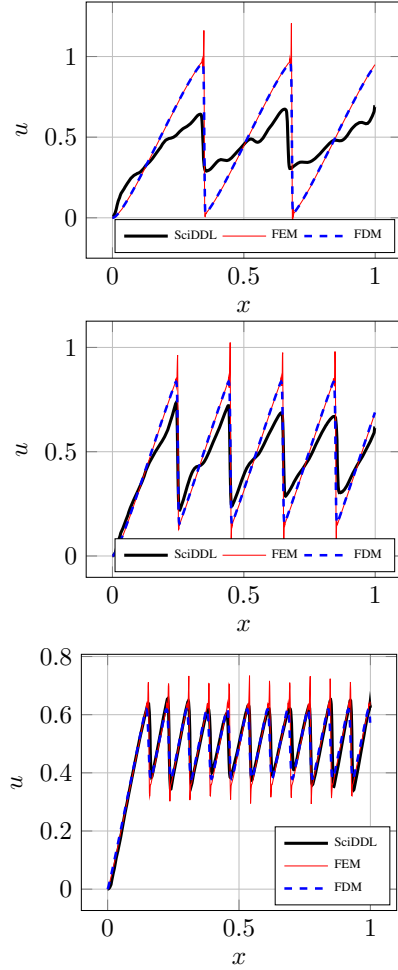


Fig. 18: The solution profile u vs. x at $t = 0.2$, compared for different methods. Resolution = 512×512 and c values are 3 (top), 5 (middle) and 13 (bottom)