Distributed Multigrid Neural Solvers on Megavoxel Domains

Aditya Balu*
Iowa State University
Ames, Iowa, USA

Sergio Botelho*
Rocket ML Inc.
Portland, Oregon, USA

Biswajit Khara*
Iowa State University
Ames, Iowa, USA

Vinay Rao
Rocket ML Inc.
Portland, Oregon, USA

Soumik Sarkar
Iowa State University
Ames, Iowa, USA

Chinmay Hegde
New York University
New York City, New York, USA

Adarsh Krishnamurthy
Iowa State University
Ames, Iowa, USA

Santi Adavani
Rocket ML Inc.
Portland, Oregon, USA

Baskar Ganapathysubramanian
Iowa State University
Ames, Iowa, USA

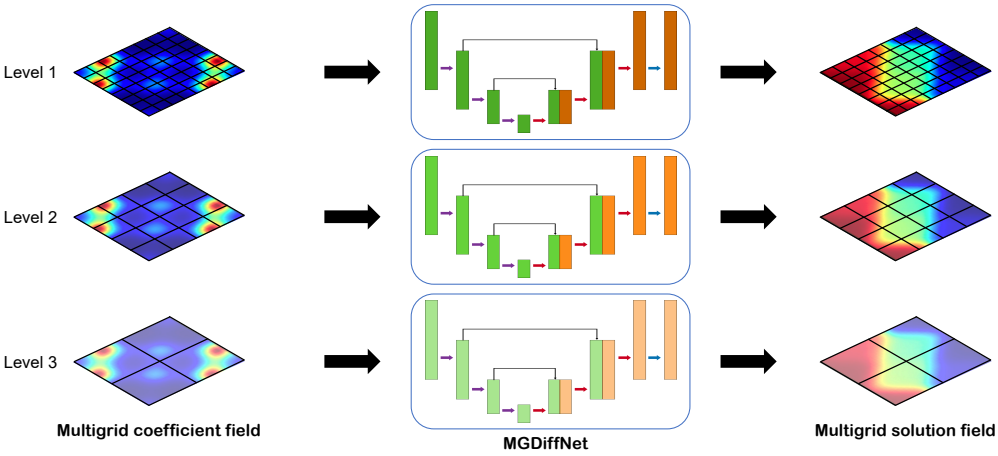


Figure 1: *We demonstrate a distributed multigrid strategy to train a neural solver that maps a coefficient field with solution field for a given parametric PDE. Coefficient fields at different multigrid resolutions are input to the same underlying network architecture at different stages of training to train the architecture at the highest resolution.*

Abstract

We consider the distributed training of large scale neural networks that serve as PDE (partial differential equation) solvers producing full field outputs. We specifically consider neural solvers for the generalized 3D Poisson equation over megavoxel domains. A scalable framework is presented that integrates two distinct advances. First, we accelerate training a large model via a method analogous to the multigrid technique used in numerical linear algebra. Here, the network is trained using a hierarchy of increasing resolution inputs in sequence, analogous to the 'V', 'W', 'F' and 'Half-V' cycles used in multigrid approaches. In conjunction with the multi-grid approach, we implement a distributed deep learning framework which significantly reduces the time to solve. We show scalability of this approach on both GPU (Azure VMs on Cloud) and CPU clusters (PSC Bridges2). This approach is deployed to train a generalized 3D Poisson solver that scales well to predict output full field solutions up to the resolution of $512 \times 512 \times 512$ for a high dimensional family of inputs. This strategy opens up the possibility of fast and scalable training of neural PDE solvers on heterogeneous clusters.

Keywords

Physics aware neural networks, Distributed training, Multigrid, Neural PDE solvers

*Authors contributed equally to the paper

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11.
https://doi.org/10.1145/3458817.3476218

ACM Reference Format:
Aditya Balu, Sergio Botelho, Biswajit Khara, Vinay Rao, Soumik Sarkar, Chinmay Hegde, Adarsh Krishnamurthy, Santi Adavani, and Baskar Ganapathysubramanian. 2021. Distributed Multigrid Neural Solvers on Megavoxel Domains. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3458817.3476218

1 Introduction

In recent years, several data-driven [42, 46] and data-free [15, 19, 22, 34, 36, 39, 43, 45, 49] approaches for solving partial differential equations (PDEs) have been proposed. The backbone of these approaches is the use of (deep) neural networks, which have proven to be capable of learning complex non-linear relationships between the inputs and the outputs. For a subset of these neural PDE solver approaches, the intent is to obtain field predictions, which can then be used to fill in a sparse amount of observable data [6, 38] or optimize the input parameters for inverse design [9, 30]. The motivation behind training such networks is to have a fast surrogate model that can quickly provide full-field solutions at a much lower cost than traditional numerical simulators. This approach is especially useful in computational design optimization, where hundreds (or thousands) of simulations are necessary to obtain an optimal design, making it computationally expensive or impractical to use traditional scientific simulators. While reduced-order modeling approaches exist for performing such design optimization, they do not necessarily capture the complete complex relationship of the underlying physics. Specifically, for design optimization at very high resolutions, reduced-order modeling may not capture the fine-scale features driving the design figure of merits (for instance, initiation of combustion instabilities).

A canonical application that motivates this work is (real-time) field reconstruction from sparse measurements. For instance, a neural PDE solver that produces high-resolution outputs of contaminant spread under diverse release scenarios can be used to assimilate sparse measurements of contaminants from sensors, and evaluate containment and evacuation strategies in real-time as well as identify the sources of the contaminant [13]. Here, both the speed of prediction and the spatio-temporal resolution are critical. While well-trained neural PDE solvers are ideal for this application, (offline) training of such high-resolution solvers is computationally expensive. This is the motivation for the current work, where we explore the idea of using neural PDE solvers to obtain the field solutions for parametric PDEs at a very high spatial resolution. More generally, the multigrid strategy affords an elegant and cost-effective approach to network architecture search (NAS) [12] since the cost of each network training is significantly reduced (see ***Remark 1***).

A large fraction of neural solvers are designed for pointwise prediction, i.e., the networks in these cases take as input a vector $\underline{x}$ of locations in the spatial domain D, and produces an output vector $\underline{u}$, by calculating the value of u at each point. They exploit the ideas of automatic differentiation [37] to solve the PDE by minimizing the residual over a set of sampled points $\underline{x}$. Due to this implicit representation, these methods do not require a mesh and rely on collocating points from the domain randomly. Apart from minimizing the volumetric residual, these approaches also satisfy the prescribed boundary conditions. Some of these methods satisfy/apply the boundary conditions exactly [25, 28, 32], while others do that in an approximate (weak) sense [26, 39, 45]. While the state-of-the-art methods mentioned here show great promise in mapping the complex non-linear relationship between the domain and the field values representing the physics, these methods have the following limitations:

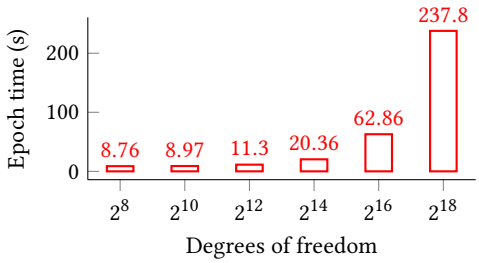


Figure 2: *Time taken per epoch for performing training at different resolutions of the 2D solution field using same network architecture.*

(1) **Non-intuitive weights and hyper-parameters:** The methods that approximately satisfy the boundary conditions do so by adding a loss function with respect to the specified boundary conditions. However, the losses have to be carefully weighed, making this a non-trivial exercise in hyper parameter tuning [47]. While recent work like Variational PINN [22], neurodiffeq [7] alleviate this issue (by the exact imposition of boundary conditions, instead of another loss), these are not yet fully developed for arbitrary boundary conditions.
(2) **Single instance solution:** Most of the approaches above use an implicit representation of the domain where the input are the points $\underline{x}$ for performing the prediction. Although the implicit representation has several advantages, such as its capability to predict the fields for any arbitrary resolution of points, there are disadvantages, such as the inability to provide topological information about the geometry. Topological information is essential for developing a robust solver that can handle changing the input geometry or the input parameters. Therefore, the above methods suffer from the limitation of their applicability to a single instance of the PDE and do not solve a family of parametric PDE instances. Recent works such as SimNet [16] attempt to capture a small domain of parametric cases instead of the complete field representation of the parametric PDE.
(3) **Scalability:** Most of these approaches (although fundamentally scalable) have not been well explored in applications to 3D spatial domains due to computational costs involved in training such deep learning models. With the increase in dimensionality, there is an increase in the number of collocation points sampled (the spatial resolution). Further, enforcing boundary conditions is much more challenging (in weak enforcement of the boundary condition). Apart from these technical issues, computational issues such as the computational cost involved in training these networks are also challenging.

A limited number of efforts address these issues. For example, Liao and Ming [29] resolve application of essential boundary conditions by using Nitsche's variational formulation. Khoo et al. [23] extend efforts for solving parametric PDEs. In additiona to these mathematical developments, recent work such as Botelho et al. [3], and Yang et al. [50] enable the scalable training of models used for solving PDEs. Specifically, Yang et al. [50] demonstrates the scalability of the framework to 27,500 GPUs. However, the application of these methods in 3-dimensional spatial domains is computationally

expensive. As the spatial domain increases, traditional PINN (and its variants) need a vast number of collocation points. Similarly, in the parametric setting, using a convolutional neural network [3, 23], the voxel resolution creates computational and memory requirement challenges. For example, in Figure 2 we see that the computational time per epoch increases quadratically with the increase in the resolution of the spatial domain. These challenges persist, especially for training neural PDE solvers at scale.

Data-parallel distributed deep-learning strategies are often used to overcome memory limitations, where multiple replicas of a model are simultaneously trained to optimize a single objective function. Typically, universities and government research labs either use on-premise HPC clusters or supercomputers such as the Summit, Bridges2, Frontera, and Stampede2. In this paper, we use a distributed deep learning strategy for performing our training on the Bridges2 cluster running on CPU nodes. However, most of these systems have very few GPU nodes (except for Summit, having 27,360 GPUs). Therefore, we use the Microsoft Azure on-demand HPC virtual machines for performing our distributed experiments on the GPU. This is especially topical, given recent efforts by federal agencies (like the US NSF) for providing cloud access via the CloudBank service.

In addition to using distributed deep learning, we also propose a new training scheme inspired by the multigrid approaches to solving PDEs. The key idea is to use a variational formulation of the loss function to train the neural network at different resolutions or levels (similar to different levels in the multigrid approach). This approach is particularly useful because the training in the lower resolutions is much faster (see Figure 2) than the training time at higher spatial resolutions. We explore strategies for efficient and scalable training of neural PDE solvers based on this approach.

Remark 1: While our PDE application motivates these developments, the distributed multigrid approach can be used to train any fully convolutional neural network that maps input fields to output fields that are resolution agnostic. This encompasses diverse applications, including semantic segmentation and image-to-image translation prevalent in computer vision.

The main contributions of this paper are:

(1) A variational loss function to solve PDEs (similar to previously proposed ideas [22, 29, 45]) but with the exact application of boundary conditions.
(2) A multigrid-inspired training scheme for training the networks at higher resolutions. We explore several multigrid training schemes and perform a detailed comparison with the direct training of the neural network at high resolutions.
(3) Cluster-agnostic data parallel distributed deep learning library to train CNNs. We illustrate versatility using on-prem and cloud based CPU and GPU HPC clusters.
(4) Demonstrated scaling of the approach to very high resolutions (up to $512 \times 512 \times 512$ voxel resolution) in 3D using CPU (on PSC Bridges2) and GPU (on Azure VMs) clusters.

The rest of the paper is arranged as follows: we first explain the mathematical preliminaries in Section 2; we explain the algorithmic contributions of our work in Section 3; we present the scaling and timing results in Section 4; and finally, we conclude and provide a few remarks on possible future work. Some of the notations and abbreviations are summarized in Table 1.

Table 1: *Notations used in paper.*

PDE	partial differential equation
FEM	finite element method
NN	neural network
MG	multigrid
GMG	geometric multigrid
SGD	stochastic gradient descent
p	number of MPI tasks in the MPI communicator
N	elements along a dimension in an FEM discretization
N_s	number of samples used in a neural optimization
N_b	number of minibatches in the optimization process
b_s	batch-size for SGD based optimization
N_w	number of neural network model parameters
G_{nn}	the neural network as a function

2 Mathematical preliminaries

2.1 Convolutional Neural Networks (CNNs)

A deep neural network consists of several layers of connections forming one network, which takes an input γ_{in} and produces an output γ_{out}. Each connecting layer (l_i) in the network can be represented as $\gamma_{l_{i+1}} = \sigma(W_{l_i} \cdot \gamma_{l_i} + b_{l_i})$, where $\sigma(...)$ represents a non-linear activation function, W_{l_i} and b_{l_i} are the weights and biases in the connection. A convolution connection (compared to a dense connection) provides a more efficient and compact connection especially for images and fields. The convolution operation ($\otimes$) between a $3D$ input representation γ and a corresponding $3D$ weight, W is given by

$$W[m,n,p] \otimes \gamma[m,n,p] = \sum_{i=-h}^{i=h} \sum_{j=-l}^{j=l} \sum_{k=-q}^{k=q} W[i,j,k] \cdot \gamma[m-i, n-j, p-k] \tag{1}$$

A series of convolutional connections, non-linear activations, and pooling forms a CNN. CNNs are more prevalent in deep learning due to their efficacy in capturing the topological information in datasets such as images, videos, voxels, etc. Several recent papers have utilized such neural networks for producing field predictions [35, 40, 51, 52]. In the next section, we provide details of the network used in this paper. Now, we shall cover some preliminaries for solving PDEs using neural networks.

2.2 DiffNet: Solving PDEs using CNNs

Consider a bounded open (spatial) domain $D \in \mathbb{R}^n, n \geq 2$ with a Lipschitz continuous boundary $\Gamma = \partial D$. We will denote the domain variable as $\underline{x}$, where the underbar denotes a vector or tuple of real numbers. In $\mathbb{R}^n$, we have $\underline{x} = (x_1, x_2, \ldots, x_n)$; but for 2D and 3D domains, we will use the more common notation $\underline{x} = (x, y)$ and $\underline{x} = (x, y, z)$ respectively. On this domain D, we consider an abstract PDE on the function $u : D \to \mathbb{R}$ as:

$$\mathcal{N}[u; s(\underline{x}, \omega)] = f(\underline{x}), \quad \underline{x} \in D \tag{2a}$$

$$\mathcal{B}(u, \underline{x}) = g(\underline{x}), \quad \underline{x} \in \Gamma \tag{2b}$$

where $\mathcal{N}$ is a differential operator (possibly nonlinear) operating on a function u. The differential equation also depends on the data of the problem s which in turn is a function of the domain variable

$\underline{x}$ and parameter ω. Thus $\mathcal{N}$ is essentially a family of PDE's parameterized by ω. $\mathcal{B}$ is a boundary operator acting on u. In general, there can be multiple boundary operators for different parts of the boundary Γ.

Given such a PDE along with appropriate boundary conditions, the goal is to find a solution u that satisfies Equation 2 as accurately as possible. Previous works [25, 39, 45] seek to find this exact mapping $u : D \rightarrow \mathbb{R}$. But as we present in the next section, we do not have to restrict ourselves to this mapping, and in fact, with the help of deep neural networks coupled with numerical methods, we can find other mappings to retrieve a discrete solution.

In this work, without any loss of generality, we focus on the Poisson equation with both Dirichlet and Neumann conditions applied on the boundaries.

2.2.1 Poisson Equation: Consider the equation:

$$-\underline{\nabla} \cdot (\nu(\underline{x})\underline{\nabla} u) = f(\underline{x}) \text{ in } D \tag{3}$$

along with the boundary conditions

$$u = g \text{ on } \Gamma_D \tag{4}$$

$$\frac{\partial u}{\partial n} = h \text{ on } \Gamma_N \tag{5}$$

where ν is the *permeability* (or *diffusivity*), f is the forcing; Γ_D and Γ_N are the boundaries of the domain D where Dirichlet and Neumann conditions are specified respectively. We will assume that $\partial D = \Gamma = \Gamma_D \cup \Gamma_N$. We are mostly interested in a steady-state mass (or heat) transfer through an inhomogeneous medium (material), which means that the material has different properties at different points. The only material property appearing in the Poisson's equation (3) is $\nu(\underline{x})$, thus the inhomogeneity can be modeled by a spatially varying ν, i.e., $\nu = \nu(\underline{x})$.

Without loss of generality, we consider the following equation:

$$-\underline{\nabla} \cdot (\tilde{\nu}(\underline{x})\underline{\nabla} u) = 0 \text{ in } D \tag{6}$$

with the boundary conditions

$$u(0, y) = 1 \tag{7}$$

$$u(1, y) = 0 \tag{8}$$

$$\frac{\partial u}{\partial n} = 0 \text{ on other boundaries} \tag{9}$$

where D is a hypercube domain in $\mathbb{R}^n$, $n = 2, 3$. Here the diffusivity $\tilde{\nu}$ is parametric, and is represented by the following log permeability expression, typically used in geological simulations and in uncertainty quantification:

$$\tilde{\nu}(\underline{x}; \omega) = \exp\left(\sum_{i=1}^{m} \omega_i \lambda_i \xi_i(x) \eta_i(y)\right) \tag{10}$$

where ω_i is an m-dimensional parameter, λ is a vector of real numbers with monotonically decreasing values in order; and ξ and η are functions of x and y respectively. We take $m = 4$, $\underline{\omega} = [-3, 3]^4$ and $\lambda_i = \frac{1}{(1+0.25a_i^2)}$, where $\underline{a} = (1.72, 4.05, 6.85, 9.82)$. Also $\xi_i(x) = \frac{a_i}{2}\cos(a_i x) + \sin(a_i x)$ and $\eta(y) = \frac{a_i}{2}\cos(a_i y) + \sin(a_i y)$.

2.3 Geometric Multigrid approach

The geometric multigrid (GMG) is a powerful tool used for scalable numerical linear algebra. The GMG approach defines a hierarchy of meshes and sequentially projects and solves the PDE on these meshes. The advantage of GMG lies in accessing the different regions of the error spectrum of a numerical operator by projecting the error on meshes of varying refinement. This is a powerful concept that can be naturally extended to training CNNs. We provide a brief outline of the major ideas of multigrid approaches below. Detailed discussions can be found in texts such as [4, 5, 14].

Suppose we want to solve the Poisson equation on a $[0, 1]^2$ domain using $N \times N$ "finite elements". This $N \times N$ grid marks the "finest" grid, which is referred to as the "Level-1" (see Figure 3). Such discrete grids are capable of representing only a finite number of Fourier modes. When classical iterative methods such as under-relaxed Jacobi iteration are used on the linear system at this level, the errors corresponding to the high frequencies ($\geq N/2$) are reduced within a few iterations, but the errors corresponding to the low frequencies ($< N/2$) take several iterations to reduce, thus rendering the solve extremely slow. But this issue can be overcome simply by performing subsequent iterations on increasingly coarser grids, e.g., a grid having $(N/2 \times N/2)$ elements (i.e., "Level-2"). The maximum frequency at this level is now $\approx N/2$, and thus the errors in the frequencies above $N/4$ are reduced faster at this level. We can continue this coarsening process by going to deeper levels until a satisfactory accuracy at the lowest Fourier (frequency) mode is achieved. In Figure 3, the coarsest level shown is "Level-4". The coarsening of the grid and projection of the solution to the coarse grid is called "restriction". In a geometric multigrid method, often, the coarse grid is simply a coarser subset of the preceding grid.

Once a solution is obtained at the deepest level, it is then propagated upwards through the finer levels. Interpolating the solution from a coarser level to a finer level is called "prolongation". After prolongation, a few more iterations might be needed to smooth out some additional errors introduced during prolongation, known as "post-smoothing". The same process is followed to go back to the finest level (Level-1). This whole idea of solving the same problem on multiple grids to strategically target all the Fourier modes of the error is the essence of the multigrid approach.

One major aspect of GMG is the choice of different grid hierarchies (or GMG cycles). Figure 3 illustrates some common grid hierarchies in the multigrid approach. It is important to note that solving the system on progressively coarser grids becomes progressively cheaper. In a V-cycle hierarchy, restriction and smoothing are performed until the coarsest grid, and then the prolongation and correction are performed until one reaches the starting mesh resolution. In a W-cycle (second from left in Figure 3) restriction

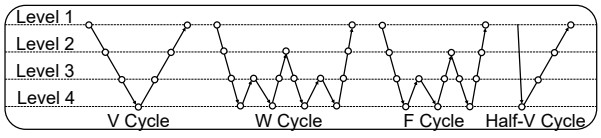


Figure 3: *Different multigrid strategies.*

and smoothing is performed to the coarsest cycle. However, instead of performing prolongation and correction to the initial mesh resolution, prolongation and correction are used alternatively to minimize the low-frequency errors and improve stability. It is important to note that this does not compromise efficiency as these alternate operations are done on really cheap coarse meshes. Subsequently, correction and prolongation are performed fully to the initial mesh resolution, just like in the V-cycle. The extra expense of the W-cycle compared to the V-cycle is progressively lower for increasing spatial dimensions [14]. The F-cycle falls somewhere between V-cycle and W-cycle in terms of expense. It starts with the restriction to the coarsest grid like the V-cycle. After reaching each level the first time, a restriction to the coarsest grid is performed in the prolongation process. In addition to the classical MG cycles discussed above, we also consider the "half-V-cycle", which is not very common in numerical analysis literature, but turns out to be a very natural method in the case of MGDiffNet (our proposed approach). It can be classified as a special case of the V-cycle, in which no smoothing is done before the coarsest grid layer. This is discussed in more detail in Section 3.1.2.

Several works have been performed in the context of multigrid approaches to deep learning [8, 21, 48] and deep learning approaches to improve multigrid operations [17, 20, 31, 33]. Here, we leverage the multigrid hierarchy and try to establish a mapping between the domain and the solution using a CNN on every grid layer. However, careful scaling and timing analysis is required to determine the best strategy, which we perform in Section 4.

3 Algorithmic Developments

3.1 Multigrid Approaches

We seek a mapping between the input s and the full field solution u in the discrete spaces. S^d denotes the discrete representation of the known quantity s. S^d could be either available only at discrete points (perhaps from some experimental data). In many cases, s is in a functional form, and thus S^d will be the values of s evaluated at the discrete points. Therefore, if we denote a MGDiffNet network by G_{nn}, then G_{nn} takes as input a discrete or functional representation of s and predicts a discrete solution field U^d_θ, where θ denotes the network parameters. For example, if we consider a PDE defined on a 2D bounded domain, G_{nn} takes a 2D matrix containing the values of s and predicts the solution field U^d_θ which is also a 2D matrix (as illustrated in Fig. 1). The weights of the network G_{nn} are initialized randomly in the beginning, and using optimization schemes, we obtain the network parameters θ, which maps the input coefficients field s to solution field u. The first step is designing the loss function based on the finite element method (FEM).

3.1.1 FEM Loss: The FEM loss involves the weakening of the PDE using an appropriate weighting functions. Let the set $\underline{X} = (\underline{x}_1, \underline{x}_2, .., \underline{x}_N) \in \mathbb{R}^{n\times N}$ denote a collection of points in $\mathbb{R}^n$ that produces a (uniform) discretization of D with a set of non-overlapping elements denoted by Q_i, $i = 1, 2, \ldots, n_{el}$ such that $\cup_i^{nel} Q_i = D$. we define $S_i = s(\underline{x}_i)$ and U_i an approximation of the unknown $u(\underline{x}_i)$. The unknown solution can be approximated as:

$$u^h_\theta = \sum_{i=1}^{N} \phi_i(\underline{x})(U_i)_\theta \tag{11}$$

where ϕ_i are the finite element basis functions.

This approximation is plugged into the PDE, after which we invoke Galerkin's method. We multiply the PDE with a test function and reduce the differentiability requirement on u^h using integration by parts:[1]

$$\int_\Omega v \left[\mathcal{N}(u^h_\theta; s) - f\right] d\underline{x} = 0 \; \forall v \in V, \tag{12}$$

which results in this following (standard FEM) form

$$B(v, u^h_\theta) - L(v) = 0 \; \forall v \in V, \tag{13}$$

where $B(v, u^h_\theta)$ is the bilinear form that encodes the PDE, while $L(v)$ is the linear form that encodes the load and the boundary conditions. By choosing the test function to be the (unknown) solution, u^h_θ, we get an energy functional whose minima is the solution:

$$J(u^h_\theta) = \frac{1}{2} B(u^h_\theta, u^h_\theta) - L(u^h_\theta). \tag{14}$$

This energy functional accounts for the PDE as well as all Neumann (and Robin) boundary conditions. This energy functional also serves as our loss function.

3.1.2 Multigrid Training of MGDiffNet: We first define the neural network, G_{nn}, to be a fully convolutional neural network with the following properties:

(1) The connections between each layer only use convolution (and/or transpose convolution) operations;
(2) The downsampling (performed using max-pooling or convolution with stride > 1) is always a factor of two;
(3) Appropriate padding is performed to ward off fence effects.

Constructing such a fully convolutional network is not difficult. A standard fully convolutional neural network, called U-Net [10, 41], satisfies all the requirements mentioned above. The primary use of such a fully convolutional neural network is that the network architecture remains the same for different input resolutions. Recall that the filter weights W for a convolution operation is not dependent on the input resolution (N in each dimension) and the same filter weights can be used to extract local information from any resolution. This means that for learning a smooth solution field, we can perform training of G_{nn} at different resolutions where the network's parameters learn the mapping between the solution field u and the coefficients field s[2].

The core idea behind different multigrid strategies is the transfer learning between two grid resolutions. Due to the fully convolutional nature of the neural network, once G_{nn} is trained at one resolution, the forward pass of the coefficients through the network itself becomes an excellent starting point for performing interpolation (prolongation) and solving the PDE at a higher resolution.

[1] For completeness, we assume $u^h_\theta \in V \subset H^1(D)$ where $H^1(D)$ denotes the Hilbert space of functions on D that have square-integrable first derivatives.

[2] Interestingly, while writing this paper, we came across work that hypothesized deep mathematical connections between numerical methods and neural nets [1], with a specific call out to a link between multigrid approaches with U-Net architectures. Our work anecdotally validates these assertions.

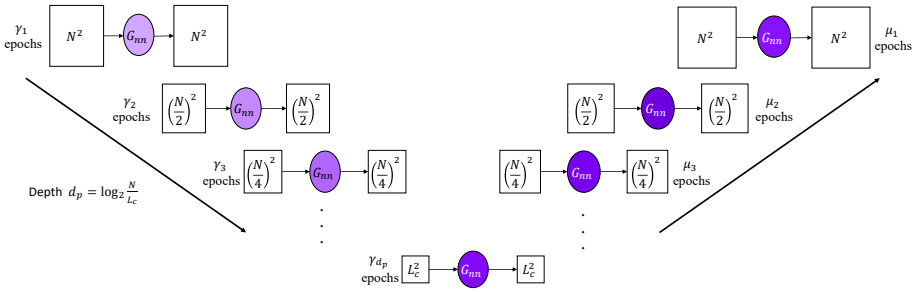


Figure 4: *Schematic diagram of a typical V-cycle in MGDiffNet for a $N \times N$ 2D problem. L_c is the grid size in one dimension at the coarsest level. Here we assume that the ratio $\frac{N}{L_c}$ is a perfect power of 2 and thus the level-depth $d_p = \log_2\left(\frac{N}{L_c}\right)$ is an integer.*

Algorithm 1 Underlying training algorithm for MGDiffNet at *each MG-level*

procedure TRAINING(G_{nn}, S^d, α, TOL)
  for epoch ← 1 to max_epoch **do**
    for mb ← 1 to max_mini_batches **do**
      Sample S^d_{mb} from the set
      $(U^d_\theta)_{int,mb} \leftarrow G_{nn}(S^d_{mb})$
      ▷ "int" stands for interior nodes
      $(U^d_\theta)_{mb} \leftarrow (U^d_\theta)_{int,mb}\chi_{int} + (U^d_\theta)_{bc}\chi_b$
      $loss_{mb} = L(U^d_\theta)$
      $\theta \leftarrow optimizer(\theta, \alpha, \nabla_\theta(loss_{mb}))$ ▷ G_{nn} is updated
    end for
  end for
  Return G_{nn}
end procedure

Algorithm 2 MGDiffNet algorithm for a V-cycle

Require: Finest Grid size N, Coarsest grid size Lc, Set of optimization inputs at each level = OPT
procedure MG(N, Lc, OPT, G_{nn}) ▷ N=grid size
  Create samples S^d for size N
  Unpack α and TOL from OPT for this level
  $G_{nn} \leftarrow$ TRAINING(G_{nn}, S^d, α, OPT) ▷ "smoothing" at this level using Algorithm 1
  if N/2 < Lc **then**
    Return G_{nn}
  else
    $G_{nn} \leftarrow$ MG(N/2, Lc, OPT, G_{nn})
  end if
  $G_{nn} \leftarrow$ TRAINING(G_{nn}, S^d, α, OPT) ▷ "post-smoothing" at this level using Algorithm 1
  Return G_{nn}
end procedure

We now train the network until convergence (defined by the early stopping criteria) to proceed to higher resolutions. In the context of deep learning, these cycles help the network become robust to different resolutions and can learn the unique mapping at all the resolutions. Here, we note that this is only true when the network learning capacity is infinite. Different filters of the convolution operation learn neighborhood information at different scales of the multigrid, thus solving the PDE faster.

Different multigrid cycles can be performed using MGDiffNet:

- **V-cycle:** The simplest strategy is the V-cycle (see Figure 3). We first run training of G_{nn} at the coarsest level (with resolution $N \times N$) for γ_1 epochs. Then, we change the input-output grid for the same problem to $N/2 \times N/2$ keeping the weights and biases learnt from Level-1. Now, we perform training on this problem for γ_2 epochs (usually $\gamma_2 > \gamma_1$). We continue this process till the deepest level (coarsest grid). Now, we go back to the previous level and let training run for a few epochs to fine-tune. The complete training algorithm is shown in Algorithm 2 and Figure 4.
- **W,F-cycles:** The W and F-cycles are performed in a similar manner as the V-cycle, except that the sequence of grids is different (see Figure 3).
- **Half-V cycle:** The "half-V cycle", an additional method available in MGDiffNet, is not common in the numerical analysis literature. Here, we actually start from a coarse grid (instead of a fine grid). Now this coarse scale model can serve as an starting point for higher resolution grids, thus obviating the need to start from scratch for a higher resolution problem and consequently saving time and resources. We show examples of this in Section 4.

In this study, we only consider one 'cycle' of multigrid. While it is certainly possible to extend this for several 'cycles' of multigrid and with more variations on which cycle to apply at which stage of the training, we restrict ourselves to just one cycle where each step of the cycle involves longer training time for several epochs. This avoids the problem of moving target (often quoted in relationship with reinforcement learning) where the distribution (or the frequencies of information) of data learned keeps changing, not allowing the network to be properly trained. Further, while the study can be performed at any arbitrary number of multigrid levels, we restrict ourselves to a maximum of 4 levels. Further, all the multigrid prolongation steps are until we reach convergence (defined using an early-stopping criterion). At the same time, all the restriction steps are trained for a fixed number of epochs (because convergence is not necessary at the higher resolutions in the beginning). Now, we will discuss our distributed data-parallel deep learning implementation.

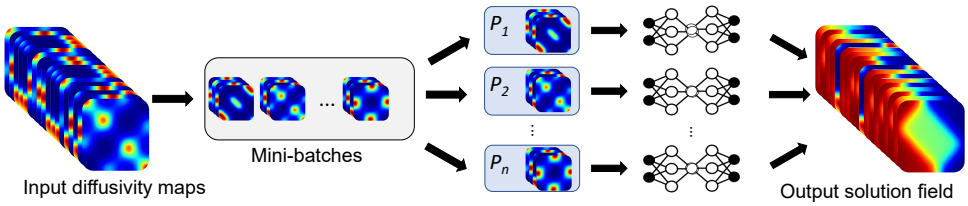


Figure 5: *Data-parallel distributed deep-learning: multiple replicas of the model are asynchronously trained by workers, each processing a local subset of the global mini-batch.*

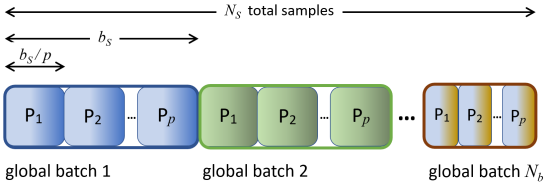


Figure 6: *Data splitting across workers in a parallel run: local mini-batches are guaranteed to always have identical sizes at any given time, promoting optimal load balance.*

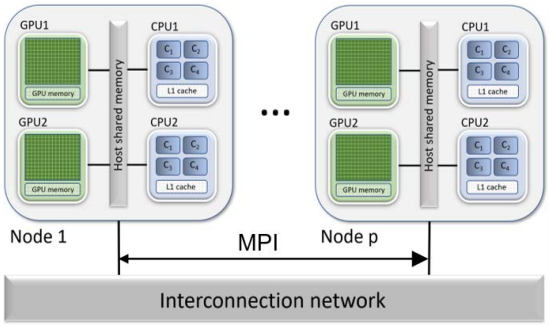


Figure 7: *Process-to-process hybrid distribution paradigm: processes communicate via MPI and spawn local threads that exploit intra-node parallelism.*

3.2 Distributed Deep Learning

One of the most widely used techniques for performing distributed deep-learning training is the *data parallel* strategy, in which identical copies of the model are simultaneously trained by independent processes that work together to minimize a common objective function [2]. For this to be possible, the training data samples (and their corresponding labels in supervised learning) must be equally split among the workers. Since stochastic optimization-based training already entails splitting the data into mini-batches, this means one has to further split the mini-batches into *local* mini-batches, which are then asynchronously processed via forward and back-propagation steps. Local gradients are computed by each worker and collectively averaged using an *all-reduce* operation. Once each worker possesses the global gradient vector, they invoke the optimizer to update their local network parameters, which are now in sync with every other worker (see Figure 5).

However, we must ensure that results are independent of the number of workers utilized, an essential tenet of high-performance computing. To accomplish that, we start by augmenting the dataset to make the total number of training samples N_s divisible by the number of workers p. Then, each global mini-batch of size b_s is divided into p equal parts, which become the local mini-batches to be dispatched to the p workers, as shown in Figure 6. This ensures that the union of the n^{th} local mini-batches across all workers will be identical to the n^{th} (global) mini-batch of the corresponding single-processor run,

$$\bigcup_{i=0}^{p} (\text{LMB})_n^i = (\text{GMB})_n \tag{15}$$

for all $n \in [0, N_b]$, where $N_b = \lceil N_s/b_s \rceil$ is the number of mini-batches in each training epoch. Module rounding errors during gradient communication, the above scheme thus guarantees that the solution will be independent of the number of workers. It also follows from the arithmetic that, for any global mini-batch size b_s chosen, the local mini-batches processed by workers at any given time will have the same size, thus optimizing load balance.

Our parallelization strategy leverages both distributed-memory MPI-based communication primitives, which handle data transfer across processes, and shared-memory OpenMP or CUDA-based multi-threading, which exploits parallelism within a node. This combination of shared memory and message-passing paradigms within the same application is known as *hybrid programming* [11], and is illustrated in Figure 7. In the specific case of our deep-learning software, MPI collective *all-reduce* calls are invoked to handle gradient communication and averaging across workers. They make use of the *ring-allReduce* algorithm [44], which has a complexity of $O(N_w + log(p))$, where N_w is the number of model parameters. Since $N_w \gg p$, we expect the communication complexity to be almost independent of the cluster size. On the other hand, the engines we use internally to execute forward and back-propagation can spawn their own Open-MP or CUDA threads, which communicate only with other threads within the same MPI process. Since MPI communication only happens outside critical multi-threaded regions, our parallelization strategy can be said to model the *process-to-process* hybrid paradigm. The number of processes launched per node and the maximum number of threads spawned by each process will depend on the specs of the cluster and details of the experiment and are chosen in such a way as to maximize resource utilization, minimize communication overhead and fulfill memory requirements.

Table 2: *Comparison between different multigrid strategies for different resolutions in 2D and 3D.*

Dimension	Resolution	Strategy	Levels	Base Time (s)	MG Time (s)	Base Loss	MG Loss	Speedup
2D	128×128	V Cycle	3	3021.05	1934.305	0.0510	0.0571	1.56×
			4		2401.070		0.0570	1.26×
		Half-V Cycle	3		3133.861		0.0568	0.96×
			4		3275.405		0.0588	0.92×
		W Cycle	3		2023.778		0.0569	1.49×
			4		2512.113		0.0597	1.20×
		F Cycle	4		2578.451		0.0584	1.17×
	256×256	V Cycle	3	9248.44	3297.706	0.0165	0.0210	2.80×
			4		3639.291		0.0209	2.54×
		Half-V Cycle	3		4585.830		0.0181	2.02×
			4		4722.950		0.0174	1.96×
		W Cycle	3		5791.277		0.0174	1.60×
			4		5597.503		0.0188	1.65×
		F Cycle	4		7401.254		0.0164	1.25×
	512×512	V Cycle	4	21860.50	10352.543	0.0050	0.0058	2.11×
		Half-V Cycle	4		11282.420		0.0053	1.94×
		W Cycle	4		10996.353		0.0062	1.99×
		F Cycle	4		17409.934		0.0053	1.26×
3D	$128 \times 128 \times 128$	Half-V Cycle	3	42422.50	7025.314	0.0400	0.0400	6.04×
	$256 \times 256 \times 256$	Half-V Cycle	4	120000.00	9000.000	0.0200	0.0200	13.33×
	$512 \times 512 \times 512$	Half-V Cycle	5	See Rem. 2	30600.000	See Rem. 2	0.0100	See Rem. 2

4 Results and Discussion

One of the key outcomes of our experiments was to demonstrate a practical approach to train MGDiffNet on domain sizes up to 512^3. We applied our framework to train MGDiffNet for resolutions up to 256^3 on GPU-based HPC clusters using on-demand multi-GPU virtual machines on **Microsoft Azure**. To train DiffNet for resolutions $> 256^3$ we used PSC **Bridges2** HPC cluster with bare-metal access to CPU nodes. We first talk about our experiments to study the multigrid approach and then the scaling studies using distributed deep learning.

4.1 Multigrid Training

We first sample the set of coefficients $\underline{\omega}$ used for generating the diffusivity maps using eq. 10. We sampled 65536 coefficients using a quasi-random Sobol sampling. The U-Net architecture used for all the experiments has a depth of 3 (i.e., a total of 3 convolution layers and 3 transpose convolution layers). First, a block of convolution and batch normalization is applied. This output is saved for later use using the skip-connection. This intermediate output is downsampled to a lower resolution for a subsequent block of convolution, batch normalization layers. This process is repeated twice. The upsampling starts where the saved outputs of similar dimensions are concatenated with the upsampling output for creating the skip-connections followed by a convolution layer. We use LeakyReLU activation for all the intermediate layers and a Sigmoid activation for the final layer. The starting filter size is 16, and we double the number of filters as the depth of the U-Net increases. For all the studies, we use the Adam optimizer [24] with a learning rate of 1×10^{-5} and the global batch size of 64.

4.1.1 Multigrid Strategies: We first study each multigrid strategy at different resolutions. In Table 2, we provide the time taken to reach convergence and the final loss. As our baseline, we perform full training at the highest resolution of the multigrid to quantify the performance. The time and the loss value at convergence for this full training are reported as Base Time and Base Loss. First, we note that all the strategies at all the resolutions converge around a similar loss value compared to the Base Loss. Also, at lower resolutions, the speedup obtained from the multigrid approaches is very marginal, and for the Half-V cycle, it is worse than the Base training time. At the same time, the V cycle has the best computational speedup.

The speedup increases with the increase in resolution for each strategy (except for the F cycle, where the increase is marginal). We also see that each strategy has a slightly different trend in speedup with the increase in resolution. To understand this, we plot the % time spent on each of the levels of resolution in Figure 8. With the primary assumption that % time spent on lower resolutions is better than that on higher resolution (based on Figure 2), we conclude that the Half-V cycle is the best. However, at lower resolutions such as 128×128, the time taken per epoch on the lower resolution is comparable with the time taken per epoch on higher resolution.

Table 3: *An analysis of cost for solving a 256^3 problem in a standalone manner and with MGDiffNet half-V cycle approach. The $/hour value provided is for a ND40rs_v2 Azure VM (Tesla V100-SXM2 with 8 GPUs per node).*

Method	Size	# epoch	Loss value	Time (min)	Cost ($/hour)	Cost ($)	Total cost ($)
MG	32	65	1.0	10	22.032	3.67	55.07
	64	30	0.1	20		7.34	
	128	20	0.04	70		25.7	
	256	10	0.02	50		18.36	
Standalone	256	400	0.02	2000	22.032	734.40	734.40

Table 4: *Network adaptation studies.*

Strategy	Base Time (s)	MG Time (s)	Base Loss	MG Loss	Speedup
Half-V Cycle (no network adaptation)	21860.50	12270.44	0.0050	0.0067	1.94×
Half-V Cycle (network adaptation)	36267.75	11803.04	0.0047	0.0052	3.07×

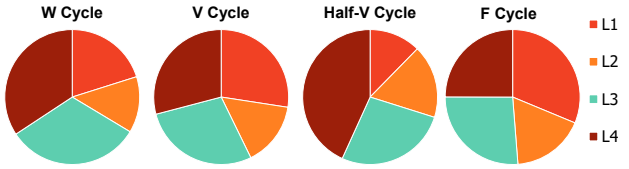


Figure 8: *Pie chart for % time spent on training at different resolutions for each multigrid strategy*

This allows for a drastic jump in speedup from 128 to 512. At the same time, the speedup for the V cycle increased and then reduced. While the speedup is desired, we want the MGDiffNet to have similar performance accuracy compared to the base network. For all resolutions, Half-V and F cycles perform closer to the Base loss, whereas the V cycle has the maximum deviation. Combined with the fact that the Half-V cycle has a much better speedup than the F cycle, we conclude that the Half-V cycle performs the best.

Table 3 provides evidence of this conclusion by showing the dollar cost on Microsoft Azure multi-GPU virtual machines. Here, a 256^3 problem is trained in two ways: first, a "standalone" training on a fixed mesh of 256^3 size; and second, through a MG half-V cycle, using a sequence of meshes of sizes 32^3, 64^3, 128^3 and 256^3. This experiment reaffirms that the MG approach outperforms the standalone training. The last column provides the total costs, and we can clearly see that the MG method reduces the computational cost by more than 10×. In the subsequent experiments, we only show results on the Half-V cycle MGDiffNet strategy.

4.1.2 Architectural Adaptation: A direct extension to the proposed multigrid approach is to adaptively add more weights for performing better at higher resolutions. This is particularly interesting when the assumption that the network has infinite learning capacity is relaxed. As soon as this assumption is relaxed, one can explore if the network learning at a lower resolution is sufficient for learning at higher resolutions. To evaluate this question, we perform an experiment where we add three additional layers (one convolutional layer and two transpose convolutional layers) and remove one learned transpose convolutional layer after training at each coarse resolution and moving to the finer resolution. The additional layers added are again initialized with random weights. However, we observe that within 20-30 mini-batches of update, the loss (which is expected to rise due to the random weights) drops down. Table 4, shows comparisons between with and without adaptation. Note that the base time and base loss for the case with architectural adaptation accounts for the final network architecture and an experiment to run full training on that final network architecture. We note that there is a marginal improvement in the loss at the same time; we show that there is a 3× improvement in training time for a very deep U-Net architecture. This ties into the theme of correlations between U-Net architecture and multigrid methods mentioned in Alt et al. [1].

4.1.3 Scaling to 3D: We next discuss training models for higher resolutions in 3D. In Table 2, we show results for $128 \times 128 \times 128$, $256 \times 256 \times 256$, and $512 \times 512 \times 512$ resolutions. Similar trends are observed for 3D problems. Here, we only show results from half-V cycle runs and compare with the base standalone training times. We point out the significant speedup achieved in the case of 128^3 ($\approx 6\times$) and 256^3 ($\approx 13\times$) resolutions. See Remark 2 for a discussion on the 512^3 case. We also show the loss performance plot of our

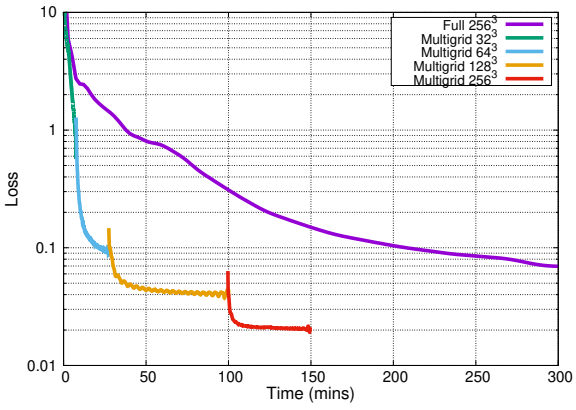


Figure 9: *Comparison of performance of base training and multigrid training for $256 \times 256 \times 256$ resolution. The multigrid strategy used here is the Half-V cycle.*

multigrid approach in comparison with full training at the same resolution in Figure 9. We see that the losses are first reduced in the lower resolutions and then further reduced at a finer resolution (as anticipated in a multigrid solver).

Remark 2: Although 3D problems as large as 256^3 can be trained on GPU nodes, a problem of size 512^3 does not fit in the GPU RAM, and we instead run across multiple CPU nodes, exploiting our MPI based library. We ran the 512^3 problem on 8 CPU nodes of Bridges-2, where one epoch takes about 6 hours. Thus, solving this problem standalone would take multiple days. Due to limited resources, we did not explore this path. That is why we do not report the standalone training time for 512^3 in Table 2. However, by using the network trained on a 256^3 grid to train the 512^3 case, we achieve convergence in just one epoch! We report this total time in the "MG Time" column (last row). This emphasizes the potential for multigrid-based deep learning methods, where an extremely expensive base case can be successfully run at a much lower cost using multigrid cycles.

4.2 Scaling to Significantly Higher Resolutions

In what follows, we demonstrate the ability to train 3D MGDiffNet on much higher resolutions by scaling out on GPU and CPU clusters with hundreds to thousands of cores. We show that we can achieve excellent speedups on both cloud and bare-metal infrastructures.

4.2.1 Scaling on a GPU Cluster: The first set of experiments were performed on a GPU cluster of NDv2-series VMs on Microsoft Azure, each containing 8 NVIDIA Tesla V100 GPUs with 32GB of memory per device. The input dataset consisted of 1024 parametric diffusivity maps of size $256 \times 256 \times 256$, as described by Equation 10. The training was performed on clusters with as many as 64 nodes (512 GPUs), using 8 devices per node for $p \geq 8$ processes (for $p < 8$, certain GPUs were left idle). The local mini-batch size was fixed at 2 since each sample required ~14GB during training, and we used the SGD-based Adam optimizer [24] (with a learning rate of 10^{-4}).

Figure 10 shows the wall-clock time per epoch, as well as the corresponding speedup. It demonstrates the ability of our distributed deep-learning solution to scale virtually linearly to 512 GPUs, reducing the runtime per epoch from 48 minutes to only 6 seconds (a speedup of 480×). Inference time (i.e., full-field prediction time) on a single GPU at this resolution was half a second.

4.2.2 Scaling on a CPU Cluster for Significantly High Resolutions: Despite achieving excellent speedups, training on GPUs is still limited by their relatively small available memory per device, which caps the maximum size of the training volumes at $256 \times 256 \times 256$. To demonstrate the ability of our software to solve problems at even higher resolutions (without implementing model-parallelism), we trained DiffNet with diffusivity maps of size $512 \times 512 \times 512$ on a cluster of AMD EPYC-7742 CPU nodes, each with 128 cores and 256GB total RAM. Figure 11 shows epoch times and speedups obtained on clusters with up to 128 nodes, with one MPI process per node (using all 128 CPU cores) and two samples per local batch. Once again, scalability is excellent up to 128 nodes. The peak memory utilization per node was 230GB, which would have been infeasible on a cluster of GPUs. The full-field prediction time on the same machine type was around 20 seconds.

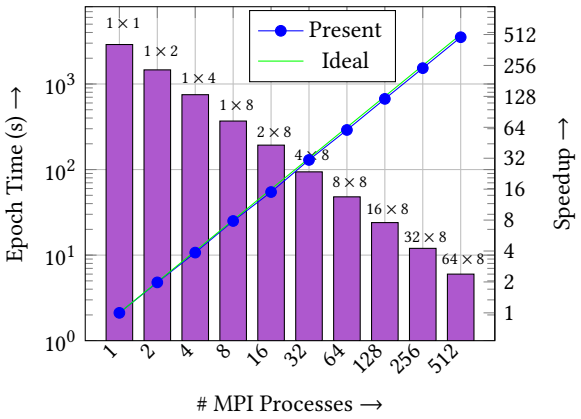


Figure 10: *Strong scaling results for training a 3D DiffNet using our distributed deep-learning framework at $256 \times 256 \times 256$ resolution on a cluster of NVIDIA Tesla V100 cloud GPUs. The labels above the bars indicate the number of nodes and the number of GPUs per node. Each epoch consists of 1024 samples.*

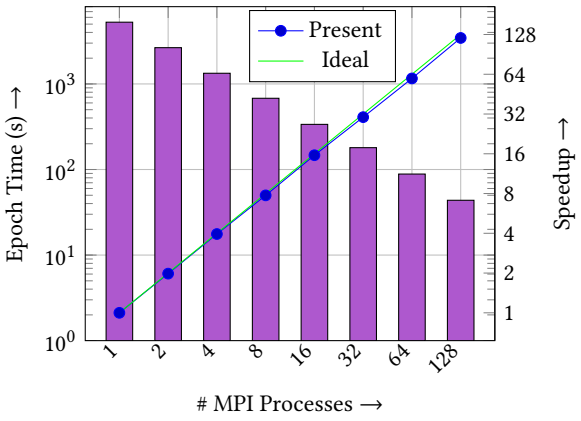
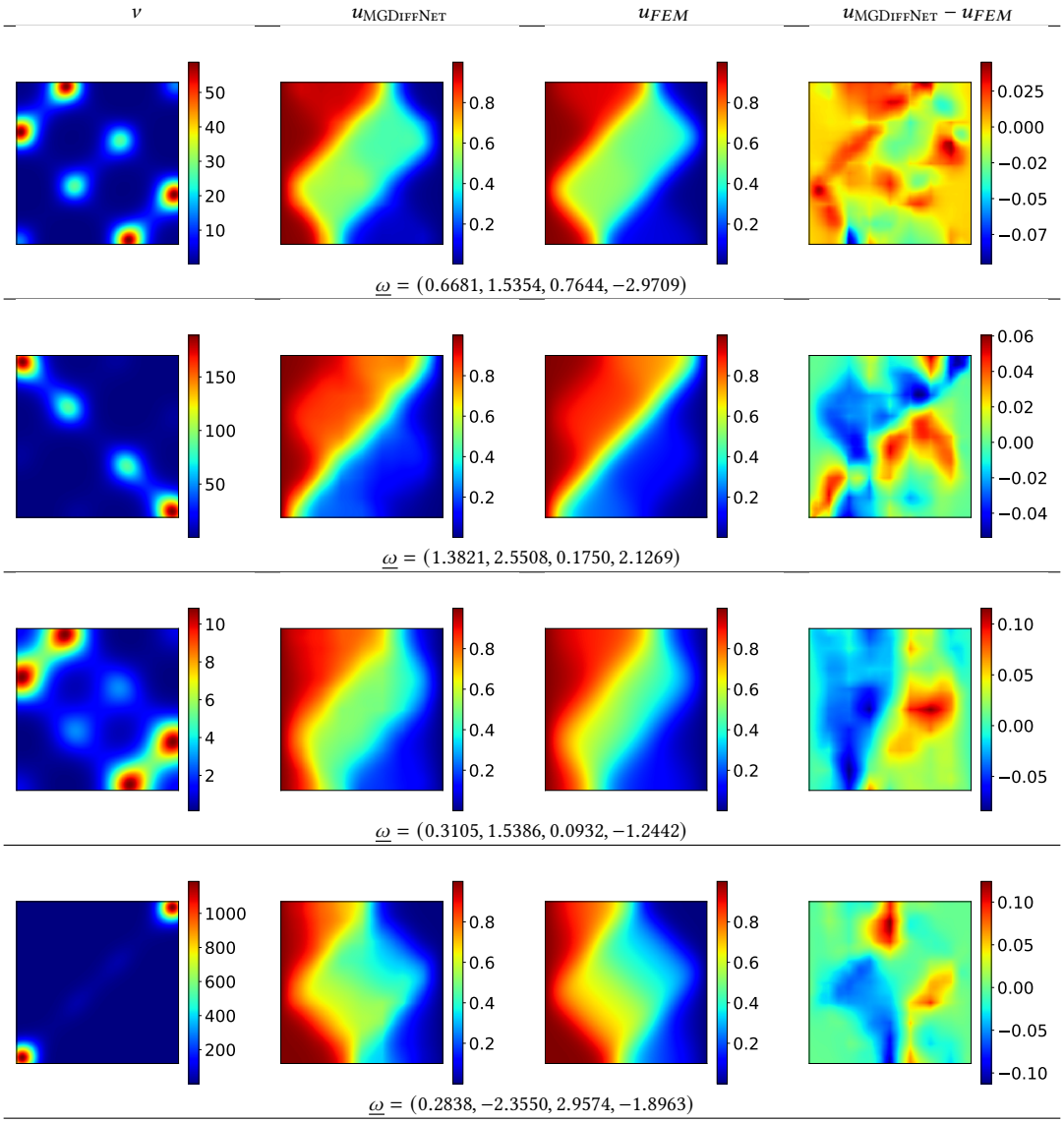


Figure 11: *Strong scaling results for training a minibatch of 3D DiffNet using our (MPI) distributed deep-learning framework at $512 \times 512 \times 512$ resolution on a cluster of AMD EPYC-7742 bare-metal nodes (with 1 MPI process per node). Each epoch consists of 128 samples.*

4.3 Comparison with Traditional FEM

We also provide some visualizations and comparisons with traditional FEM simulations for the same parameters. Here, by "traditional FEM", we mean the case where an equation is formed for each unknown in the discrete domain, and then a matrix-vector system is solved by numerical linear algebra methods [18, 27]. Table 6 shows the visualization of the predictions from the multigrid trained network for 512×512. An example in 3D is shown in Table 7. We see the MGDiffNet predicts the solution field accurately. We also compare the results obtained by different multigrid strategies to confirm that the Half-V cycle predictions are the best among all the strategies. We also show visualization of a few anecdotal solution fields produced using MGDiffNet in 2D (Table 5).

Table 5: *Visualization of MGDiffNet predictions and comparison with traditional FEM solutions for 2 anecdotal values of $\underline{\omega}$.*



Another important comparison is between the time taken for inference on MGDiffNet versus the time taken for performing one "traditional" FEM solve. The FEM simulation takes about 576 seconds for the $512 \times 512 \times 512$ resolution on 2 Stampede2 SKX nodes (this is the minimum number of nodes required to fit the FEM problem in memory). But the MGDiffNet inference, on the other hand, takes only 20 seconds on 1 Stampede2 SKX node (unlike training, inference can be performed on a single node). Since the solutions are valid for a range of PDE parameters, the impact of our framework in reducing the computational time while performing inverse design will be substantial. We also note that there is no need for any data annotation in this framework.

5 Conclusion and Future Work

In this work, we propose a distributed multigrid neural solver for solving PDEs at large spatial dimensions with efficient use of computational resources. To this end, we contribute a numerical multigrid-inspired training scheme for fully convolutional neural networks and further implement a distributed data-parallel training strategy to train networks up to a resolution of $512 \times 512 \times 512$ ($\approx 134M$ voxels). Our multigrid-based training results show a 6X speedup over the baseline full training at higher resolutions with negligible loss in performance. Further, our method scales almost linearly with minimal communication costs in a distributed environment over both CPU and GPU clusters. This approach opens up the efficient training of parametric PDEs for use in Scientific ML applications.

Table 6: *Visualization of MGDiffNet predictions with different multigrid strategies. The input* $\underline{\omega} = (0.3105, 1.5386, 0.0932, -1.2442)$

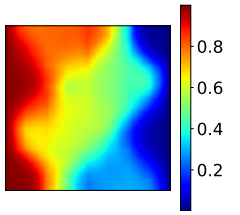
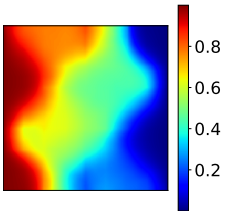
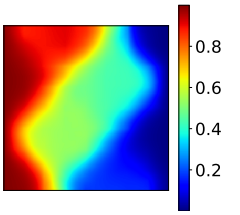
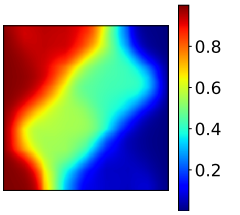
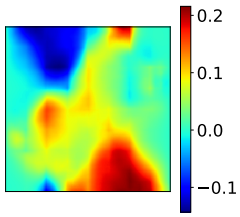
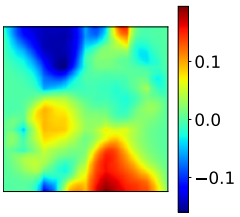
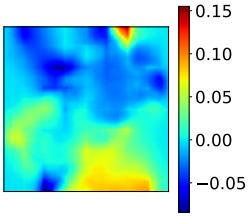
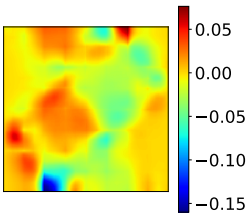
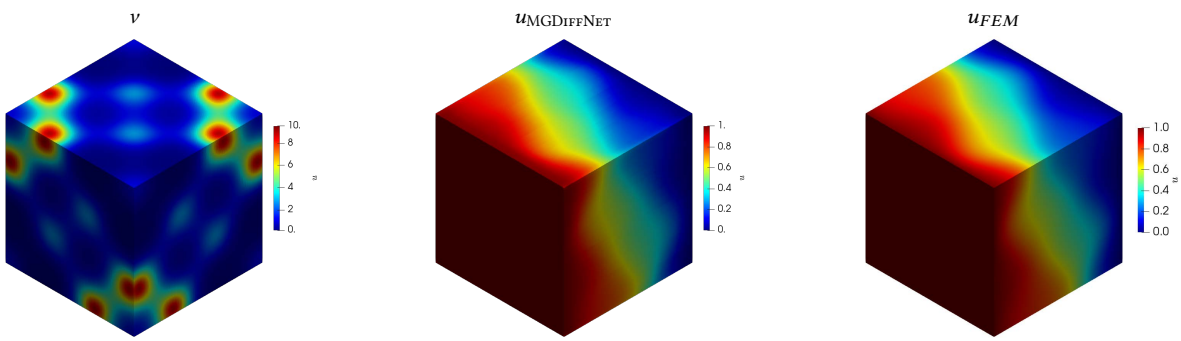
Strategy	V Cycle	W Cycle	F Cycle	Half-V Cycle
$u_{\text{MGDiffNet}}$				
$u_{\text{MGDiffNet}} - u_{FEM}$				

Table 7: *Visualization of MGDiffNet predictions and comparison with traditional FEM solutions for* $\underline{\omega} = (0.3105, 1.5386, 0.0932, -1.2442)$.



Additionally, this approach can be naturally applied to a variety of high-resolution image-to-image translation tasks.

There are several avenues of future work that follow:

- Scaling beyond megavoxels to gigavoxels. This is currently mainly limited by the CPU memory, since we estimate a single 1024^3 solve would require about $2TB$ of memory on one node. Extending our approach to allow *model-parallel* distributed deep learning could alleviate this issue.
- Elucidating the mathematical connections between the multigrid approach with stability and convergence of the training.
- Deploying this neural PDE Poisson solver for applications in topology optimization, flow through porous media, and thermal transport in composites–all of which are defined by Equation 3.
- Deploying this framework to other PDE's where having high-resolution outputs is critical for control (via model predictive control approaches).

We envision such bidirectional linkages between numerical linear algebra and scalable solutions of neural networks to significantly accelerate scientific computing workflows.

Acknowledgments

This work was partly supported by the ARPA-E DIFFERENTIATE under grant DE-AR0001215 and National Science Foundation under grants RII award number(s): 2019574, COALESCE award number(s): 1954556, CM award number(s): 1644441 and CAREER award number(s): 1750865. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by NSF grant ACI-1548562 and the Bridges2 system supported by NSF grant ACI-1445606, at the Pittsburgh Supercomputing Center (PSC). We also used Microsoft Azure compute resources for performing some of the GPU performance results shown.

References

[1] T. Alt, P. Peter, J. Weickert, and K. Schrader, "Translating numerical concepts for pdes into neural architectures," *arXiv preprint arXiv:2103.15419*, 2021.

[2] T. Ben-nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *arXiv preprint arXiv:1802.09941v2*, 2018.

[3] S. Botelho, A. Joshi, B. Khara, S. Sarkar, C. Hegde, S. Adavani, and B. Ganapathysubramanian, "Deep generative models that solve pdes: Distributed computing for training large data-free models," *arXiv preprint arXiv:2007.12792*, 2020.

[4] J. H. Bramble, *Multigrid methods*. Chapman and Hall/CRC, 2019.

[5] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A multigrid tutorial*. SIAM, 2000.

[6] S. Cai, H. Li, F. Zheng, F. Kong, M. Dao, G. E. Karniadakis, and S. Suresh, "Artificial intelligence velocimetry and microaneurysm-on-a-chip for three-dimensional analysis of blood flow in physiology and disease," *Proceedings of the National Academy of Sciences*, vol. 118, no. 13, 2021.

[7] F. Chen, D. Sondak, P. Protopapas, M. Mattheakis, S. Liu, D. Agarwal, and M. Di Giovanni, "Neurodiffeq: A python package for solving differential equations with neural networks," *Journal of Open Source Software*, vol. 5, no. 46, p. 1931, 2020.

[8] Y. Chen, B. Dong, and J. Xu, "Meta-mgnet: Meta multigrid networks for solving parameterized partial differential equations," *arXiv preprint arXiv:2010.14088*, 2020.

[9] Y. Chen, L. Lu, G. E. Karniadakis, and L. Dal Negro, "Physics-informed neural networks for inverse problems in nano-optics and metamaterials," *Optics express*, vol. 28, no. 8, pp. 11 618–11 633, 2020.

[10] Ö. Çiçek, A. Abdulkadir, S. S. Lienkamp, T. Brox, and O. Ronneberger, "3D U-Net: learning dense volumetric segmentation from sparse annotation," in *International conference on medical image computing and computer-assisted intervention*. Springer, 2016, pp. 424–432.

[11] T. V. T. Duy, K. Yamazaki, K. Ikegami, and S. Oyanagi, "Hybrid mpi-openmp paradigm on smp clusters: Mpeg-2 encoder and n-body simulation," *arXiv preprint arXiv:1211.2292*, 2012.

[12] T. Elsken, J. H. Metzen, F. Hutter *et al.*, "Neural architecture search: A survey." *J. Mach. Learn. Res.*, vol. 20, no. 55, pp. 1–21, 2019.

[13] A. D. Fontanini, U. Vaidya, and B. Ganapathysubramanian, "A methodology for optimal placement of sensors in enclosed environments: A dynamical systems approach," *Building and Environment*, vol. 100, pp. 145–161, 2016.

[14] W. Hackbusch, *Multi-grid methods and applications*. Springer Science & Business Media, 2013, vol. 4.

[15] 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.

[16] O. Hennigh, S. Narasimhan, M. A. Nabian, A. Subramaniam, K. Tangsali, M. Rietmann, J. d. A. Ferrandis, W. Byeon, Z. Fang, and S. Choudhry, "Nvidia simnet^{TM}: an ai-accelerated multi-physics simulation framework," *arXiv preprint arXiv:2012.07938*, 2020.

[17] R. Huang, R. Li, and Y. Xi, "Learning optimal multigrid smoothers via neural networks," *arXiv preprint arXiv:2102.12071*, 2021.

[18] T. J. Hughes, *The finite element method: linear static and dynamic finite element analysis*. Courier Corporation, 2012.

[19] S. Karumuri, R. Tripathy, I. Bilionis, 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.

[20] A. Katrutsa, T. Daulbaev, and I. Oseledets, "Deep multigrid: learning prolongation and restriction matrices," *arXiv preprint arXiv:1711.03825*, 2017.

[21] T.-W. Ke, M. Maire, and S. X. Yu, "Multigrid neural architectures," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 6665–6673.

[22] E. Kharazmi, Z. Zhang, and G. E. Karniadakis, "hp-VPINNs: Variational physics-informed neural networks with domain decomposition," *Computer Methods in Applied Mechanics and Engineering*, vol. 374, p. 113547, 2021.

[23] Y. Khoo, J. Lu, and L. Ying, "Solving parametric pde problems with artificial neural networks," *arXiv preprint arXiv:1707.03351*, 2017.

[24] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2015.

[25] I. E. Lagaris, A. Likas, and D. I. Fotiadis, "Artificial neural networks for solving ordinary and partial differential equations," *IEEE transactions on neural networks*, vol. 9, no. 5, pp. 987–1000, 1998.

[26] I. E. Lagaris, A. C. Likas, and D. G. Papageorgiou, "Neural-network methods for boundary value problems with irregular boundaries," *IEEE Transactions on Neural Networks*, vol. 11, no. 5, pp. 1041–1049, 2000.

[27] M. G. Larson and F. Bengzon, *The finite element method: theory, implementation, and applications*. Springer Science & Business Media, 2013, vol. 10.

[28] H. Lee and I. S. Kang, "Neural algorithm for solving differential equations," *Journal of Computational Physics*, vol. 91, no. 1, pp. 110–131, 1990.

[29] Y. Liao and P. Ming, "Deep nitsche method: Deep ritz method with essential boundary conditions," *arXiv preprint arXiv:1912.01309*, 2019.

[30] L. Lu, R. Pestourie, W. Yao, Z. Wang, F. Verdugo, and S. G. Johnson, "Physics-informed neural networks with hard constraints for inverse design," *arXiv preprint arXiv:2102.04626*, 2021.

[31] I. Luz, M. Galun, H. Maron, R. Basri, and I. Yavneh, "Learning algebraic multigrid using graph neural networks," in *International Conference on Machine Learning*. PMLR, 2020, pp. 6489–6499.

[32] A. Malek and R. S. Beidokhti, "Numerical solution for high order differential equations using a hybrid neural network—optimization method," *Applied Mathematics and Computation*, vol. 183, no. 1, pp. 260–271, 2006.

[33] N. Margenberg, C. Lessig, and T. Richter, "Structure preservation for the deep neural network multigrid solver," *arXiv preprint arXiv:2012.05290*, 2020.

[34] 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.

[35] A. G. Özbay, S. Laizet, P. Tzirakis, G. Rizos, and B. Schuller, "Poisson cnn: Convolutional neural networks for the solution of the poisson equation with varying meshes and dirichlet boundary conditions," *arXiv preprint arXiv:1910.08613*, 2019.

[36] 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.

[37] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017. [Online]. Available: https://openreview.net/forum?id=BJJsrmfCZ

[38] 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.

[39] M. Raissi, P. Perdikaris, and G. E. 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.

[40] R. Ranade, C. Hill, and J. Pathak, "Discretizationnet: A machine-learning based solver for navier–stokes equations using finite volume discretization," *Computer Methods in Applied Mechanics and Engineering*, vol. 378, p. 113722, 2021.

[41] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*. Springer, 2015, pp. 234–241.

[42] S. Rudy, A. Alla, S. L. Brunton, and J. N. Kutz, "Data-driven identification of parametric partial differential equations," *SIAM Journal on Applied Dynamical Systems*, vol. 18, no. 2, pp. 643–660, 2019.

[43] E. Samaniego, C. Anitescu, S. Goswami, V. M. Nguyen-Thanh, H. Guo, K. Hamdia, X. Zhuang, and T. Rabczuk, "An energy approach to the solution of partial differential equations in computational mechanics via machine learning: Concepts, implementation and applications," *Computer Methods in Applied Mechanics and Engineering*, vol. 362, p. 112790, 2020.

[44] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018.

[45] 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.

[46] J. Tompson, K. Schlachter, P. Sprechmann, and K. Perlin, "Accelerating eulerian fluid simulation with convolutional networks," in *International Conference on Machine Learning*. PMLR, 2017, pp. 3424–3433.

[47] R. van der Meer, C. Oosterlee, and A. Borovykh, "Optimally weighted loss functions for solving pdes with neural networks," *arXiv preprint arXiv:2002.06269*, 2020.

[48] C.-Y. Wu, R. Girshick, K. He, C. Feichtenhofer, and P. Krahenbuhl, "A multigrid method for efficiently training video models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 153–162.

[49] L. Yang, D. Zhang, and G. E. Karniadakis, "Physics-informed generative adversarial networks for stochastic differential equations," *arXiv preprint arXiv:1811.02033*, 2018.

[50] L. Yang, S. Treichler, T. Kurth, K. Fischer, D. Barajas-Solano, J. Romero, V. Churavy, A. Tartakovsky, M. Houston, M. Prabhat *et al.*, "Highly-scalable, physics-informed gans for learning solutions of stochastic pdes," in *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE, 2019, pp. 1–11.

[51] Y. Zhu and N. Zabaras, "Bayesian deep convolutional encoder–decoder networks for surrogate modeling and uncertainty quantification," *Journal of Computational Physics*, vol. 366, pp. 415–447, 2018.

[52] Y. Zhu, N. Zabaras, 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.

A Artifact Description

Summary of Reported Experiments

We performed the experiments (all experiments are described in the "Results and Discussions" section of the paper):

(1) Comparison of strategies - these were done on Azure cloud platform.
(2) Scaling studies were performed for training MGDiffNet of $256 \times 256 \times 256$ and lower were performing on Azure cloud platform and studies above $256 \times 256 \times 256$ were performed on PSC Bridges2.
(3) Solving the PDE using FEM for comparison with MGDiffNet results was done on PSC Bridges2 using 1 Regular Memory node.

Modules loaded on Bridges2 for MGDiffNet experiments:

```
1) cmake/3.16.1
2) gcc/10.2.0
3) openmpi/4.0.5-gcc10.2.0
```

Libraries Dependencies

The following dependencies are required to compile the code:

- C/C++ compilers with C++11 standards and OpenMP support
- MPI implementation (e.g. openmpi, mvapich2)
- Petsc 3.8 or higher
- ZLib compression library (used to write `.vtu` files in binary format with compression enabled)
- MKL / LAPACK library
- CMake 2.8 or higher version
- OpenCV 3.4.2

Computing Configuration

Relevant computational hardware details are provided here:

Table 8: *Functional specifications of Microsoft Azure and Bridges2 infrastructures used in our experiments.*

Specification	Microsoft Azure	Bridges2
Type	Virtual Machine	Bare-Metal
CPU	Intel Xeon Platinum 8168	AMD EPYC 7742
CPU cores	40	128
Memory (GB)	672	256
GPU	Tesla V100	-
GPU Memory (GB)	32	-
No. of GPUs	8	-
Interconnect	EDR Infiniband	HDR Infiniband
Bandwidth	100 Gb/sec	200 Gb/sec
Topology	Fat tree	Fat tree

Artifact Availability

Software Artifact Availability: Some author-created software artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: Some author-created data artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Proprietary Artifacts: There are associated proprietary artifacts that are not created by the authors. Some author-created artifacts are proprietary.

Author-Created or Modified Artifacts:

```
Persistent ID: None
Artifact name: MGDiffNet
Citation of artifact: MGDiffNet is a proprietary
↪ software from an early-stage startup whose
↪ business model is based on software licensing.
↪ Due to this reason, we cannot disclose our
↪ software details.
```

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We performed the experiments (all experiments are described in the “Results and Discussions" section of the paper):

(1) Comparison of strategies - these were done on Azure cloud platform.
(2) Scaling studies were performed for training of $256 \times 256 \times 256$ and lower were performing on Azure cloud platform and studies above $256 \times 256 \times 256$ were performed on PSC Bridges2.
(3) Solving the PDE using FEM for comparison with results was done on PSC Bridges2 using 1 Regular Memory node.

Modules loaded on Bridges2 for experiments:

```
1) cmake/3.16.1
2) gcc/10.2.0
3) openmpi/4.0.5-gcc10.2.0
```

Author-Created or Modified Artifacts:

```
Persistent ID: None
Artifact name: MGDiffNet
Citation of artifact: MGDiffNet is a proprietary
↪  software from an early-stage startup whose
↪  business model is based on software licensing.
↪  Due to this reason, we cannot disclose our
↪  software details. The data parallel distributed
↪  deep learning library written in C++ and Python to
↪  train CNNs on CPU/GPU based HPC clusters both
↪  on-prem and on cloud. Key features of the library
↪  include a) user defined variational loss
↪  functions to solve PDEs, b) ability to use
↪  multiple resolutions along with transfer learning
↪  between resolutions in a single run, and c)
↪  ability to experiment with different multigrid
↪  approaches like V, W, F, half-V cycle.
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Intel Xeon Platinum 8168, AMD EPYC 7V12, NVIDIA Tesla V100

Operating systems and versions: Ubuntu 18.04

Compilers and versions: gcc 7.5.0

Applications and versions: OpenCV 3.4.2

Libraries and versions: OpenMPI v3.1.3, HPCX

Key algorithms: Data parallel Distributed Deep Learning