Breaking Boundaries: Distributed Domain Decomposition with Scalable Physics-Informed Neural PDE Solvers

Arthur Feeney*
Zitong Li*
afeeney@uci.edu
zitongl5@uci.edu
University of California, Irvine
USA

Ramin Bostanabad
raminb@uci.edu
University of California, Irvine
USA

Aparna Chandramowlishwaran
amowli@uci.edu
University of California, Irvine
USA

ABSTRACT

Mosaic Flow is a novel domain decomposition method designed to scale physics-informed neural PDE solvers to large domains. Its unique approach leverages pre-trained networks on small domains to solve partial differential equations on large domains purely through inference, resulting in high reusability. This paper presents an end-to-end parallelization of Mosaic Flow, combining data parallel training and domain parallelism for inference on large-scale problems. By optimizing the network architecture and data parallel training, we significantly reduce the training time for learning the Laplacian operator to minutes on 32 GPUs. Moreover, our distributed domain decomposition algorithm enables scalable inferences for solving the Laplace equation on domains 4096× larger than the training domain, demonstrating strong scaling while maintaining accuracy on 32 GPUs. The reusability of Mosaic Flow, combined with the improved performance achieved through the distributed-memory algorithms, makes it a promising tool for modeling complex physical phenomena and accelerating scientific discovery.

CCS CONCEPTS

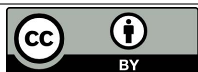
• **Computing methodologies → Neural networks**; **Distributed algorithms**; • **Mathematics of computing → Partial differential equations**.

KEYWORDS

Physics-informed machine learning, neural operators, domain decomposition, large-scale PDEs, data parallel training, scalable distributed inference

ACM Reference Format:
Arthur Feeney, Zitong Li, Ramin Bostanabad, and Aparna Chandramowlishwaran. 2023. Breaking Boundaries: Distributed Domain Decomposition with Scalable Physics-Informed Neural PDE Solvers. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3581784.3613217

*Both authors contributed equally to this research.



This work is licensed under a Creative Commons Attribution International 4.0 License.

SC '23, November 12–17, 2023, Denver, CO, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0109-2/23/11.
https://doi.org/10.1145/3581784.3613217

1 INTRODUCTION

Scientific machine learning (SciML) is an emerging field that aims to integrate scientific knowledge into the development of machine learning models. By leveraging domain expertise, SciML reduces the reliance on massive datasets that are often scarce or difficult to create in many scientific fields, such as fluid dynamics [3]. To achieve this integration, researchers have proposed various innovative strategies, ranging from incorporating scientific principles into deep neural network architectures and loss functions, developing hybrid models, using transfer learning and domain adaptation techniques, and employing Bayesian techniques [26, 40, 43, 44].

Among the above approaches, physics-informed neural networks (PINNs) have shown promise for solving complex problems that involve partial differential equations (PDEs) by incorporating physical laws and constraints [46]. By softly enforcing PDE constraints in the loss function, PINNs can learn from limited data and still provide accurate predictions. Unlike traditional methods, PINNs are mesh-free and time-continuous, making them attractive for many complex scientific applications. The growing interest in physics-informed machine learning has led to the development of numerous software libraries that offer an easy and efficient way to create PINNs. Some of the popular libraries include DeepXDE [39], NVIDIA Modulus [21], and SciANN [17].

While PINNs have shown great promise in solving problems in small domains with simple geometries, they face challenges when applied to larger domains. As the domain size increases, the complexity of the problem also grows, necessitating larger networks to capture the underlying features accurately. Since the PINN loss function can be highly non-convex, larger networks can result in a stiff and hard optimization problem, leading to significantly slower convergence with reduced accuracy or no convergence at all [29, 54]. Additionally, training PINNs for large domains require significant computational resources, which can limit their applicability to real-world problems.

Domain decomposition [9] has emerged as a potential solution to improve the scalability and convergence of neural PDE solvers on large domains. This approach involves breaking down the challenging global optimization problem on the entire domain into many smaller and simpler local optimization sub-problems. Table 1 summarizes the different domain decomposition methods (DDM) that have been developed for neural PDE solvers. They can be broadly classified into two categories. The first category is *non-overlapping DDM*, where the domain is divided into non-overlapping subdomains. A separate neural network is trained on each subdomain,

Table 1: State-of-the-art domain decomposition methods for neural PDE solvers. The overlapping approaches are based on Schwarz methods or inspired by them as denoted by ∗. Unlike MosaicFlows which solves PDEs on arbitrarily large domains using only neural network inferences, all other approaches require training a new model for each new domain.

DDM	Subdomains	PINN formulation	Interface condition	Interface resolved	Dist alg	Dist eval
DPINN [12]	non-overlapping	residuals	loss terms	training	✗	✗
XPINN [24, 49]	non-overlapping	residuals	loss terms	training	✓	✓
cPINN [25, 49]	non-overlapping	conservation laws	loss terms	training	✓	✓
hp-VPINNs [27]	non-overlapping	variational residuals	loss terms	training	✗	✗
DeepDDM [34]	overlapping	residuals	Schwarz	training	✗	✗
D3M [32]	overlapping	variational residuals	Schwarz	training	✗	✗
FBPINN [7, 42]	overlapping	residuals	Schwarz*	training	✓	✗
Mosaic Flow [53]	overlapping	residuals	Schwarz*	inference	✗	✗
Dist-MF (this paper)	overlapping	residuals	relaxed Schwarz	inference	✓	✓

and continuity across subdomain interfaces is enforced using additional interface terms in the PINN loss function. DPINN [12], XPINN [24], cPINN [25], and hp-VPINN [27] belong to this category. A key drawback of these approaches is inherent to their design in enforcing continuity across the subdomain interfaces. Since the interface conditions are only weakly constrained in the loss function, it can lead to artificial discontinuities at the subdomain interfaces. The additional interface terms not only introduce additional hyperparameters that need to be tuned to train the best model, they can also compete with the PDE losses. This contention can often slow convergence [56]. Nevertheless, they are relatively simple to implement, and cPINN/XPINN have been parallelized to scale to multiple GPUs, leading to reductions in training times [49]. The second category is *overlapping DDM*, which divides the domain into overlapping subdomains. In DeepDDM [34] and D3M [32], PINNs replace the subdomain solvers with variants of the classical Schwarz domain decomposition method [36]. FBPINN [42] employs a separate input normalization in each subdomain and summation over all subdomain networks. Continuity between interfaces is enforced via the construction of the PINN solution ansatz[1]. It can also be cast in the form of additive, multiplicative, and hybrid Schwarz methods [7]. In contrast to the above approaches that require *training separate neural PDE solvers* on non-overlapping or overlapping subdomains and resolving the interface between them, Mosaic Flow [53] solves PDEs on larger domains using *only inference*. An iterative algorithm inspired by the alternating Schwarz method updates the solution in subdomains using the pre-trained network inferences while maintaining the spatial regularity of the solution at subdomain boundaries. This eliminates the need to retrain the neural network for each new domain, making Mosaic Flow highly reusable across different domain sizes and significantly reducing computational costs.

Contributions and Findings. We propose an end-to-end parallelization pipeline for scaling Mosaic Flow to large domains that encompasses both training and inference.

- **Training on small domains.** We redesign the physics-informed neural PDE solver with focus on performance and scalability. The resulting network with an innovative input embedding and optimized architecture, combined with data-parallel training, reduces the training time for learning the Laplacian operator from hours to just 2 minutes on 32 NVIDIA A30 GPUs.
- **Inference on large domains.** To enable scalable distributed inferences using the pre-trained neural PDE solvers on arbitrarily large domains, we propose a relaxation in synchronization. The relaxed distributed algorithm maintains accuracy as shown in Figure 1 while scaling up inferences to domains 4096 times larger for solving the Laplace equation. To the best of our knowledge, this is the largest domain solved in seconds using 32 GPUs, combining DDM with physics-informed neural PDE solvers as sub-domain solvers. In addition, we demonstrate strong and weak scaling up to 32 GPUs.

[1]A solution ansatz is a mathematical form or assumption about the solution to a PDE. It captures specific properties that the solution should satisfy, such as boundary conditions or initial conditions.

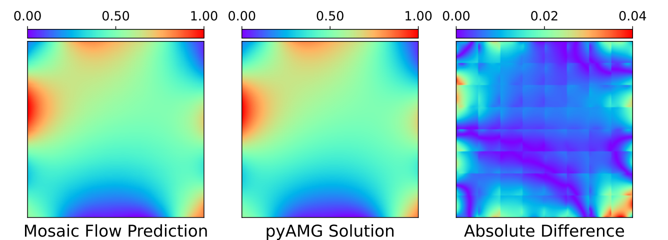


Figure 1: The leftmost sub-figure shows the solution using pyAMG to solve the Laplace equation on a $2{\times}2$ spatial domain with $128{\times}128$ resolution. The boundary condition is generated through a Gaussian process. The middle sub-figure shows the result of using distributed Mosaic Flow predictor on the same domain. The rightmost sub-figure shows the absolute difference between the two.

2 BACKGROUND

This section begins with a brief introduction to the problem definition and physics-informed neural PDE solvers. We then delve into domain decomposition and elaborate on how Mosaic Flow leverages this approach to implement large-scale physics-informed neural solvers.

2.1 Problem Definition

In this work, we develop SciML models to solve boundary value problems (BVP) of the form

$$
\begin{aligned}
D[u(\boldsymbol{x})] &= f(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega \\
B[u(\boldsymbol{x})] &= g(\boldsymbol{x}), \quad \boldsymbol{x} \in \partial\Omega
\end{aligned} \tag{1}
$$

The vectors $\boldsymbol{x}$ are in the domain Ω or on the domain boundary $\partial\Omega$. The function u is the solution of the differential equation. The differential operator is denoted by D, while B is the boundary operator. The forcing function is f, and g is the boundary function. A classic example of a BVP is the 2D Laplace equation with a Dirichlet boundary condition:

$$
\begin{aligned}
\Delta u(\boldsymbol{x}) &= 0, \quad \boldsymbol{x} \in \Omega \\
u(\boldsymbol{x}) &= g(\boldsymbol{x}), \quad \boldsymbol{x} \in \partial\Omega
\end{aligned} \tag{2}
$$

where the vector $\boldsymbol{x} = [x, y]$ and $\Delta = (\partial^2/\partial x^2 + \partial^2/\partial y^2)$ is the Laplacian operator [13].

2.2 Neural PDE Solvers

Neural PDE solvers (or neural solvers for short) [28, 35, 38] are a type of model that learns to approximate the PDE *solution operator* and solve various instances of a BVP with different boundary conditions. They are trained using a dataset that consists of solved boundary value problems for a specific PDE. A neural solver may take a discretized boundary function as input, denoted by $\hat{\boldsymbol{g}} = \{g(\boldsymbol{x}_{bc}^1), \ldots, g(\boldsymbol{x}_{bc}^N)\}$, where $\boldsymbol{x}_{bc}^i$ are N points sampled on the boundary. $\hat{\boldsymbol{g}}$ specifies the particular instance of the BVP to solve. Therefore, an neural solver, represented by the function $\mathcal{N}(\boldsymbol{x}, \hat{\boldsymbol{g}}; \theta)$, approximates the solution $u(\boldsymbol{x})$ for the instance of the BVP determined by the boundary function g.

This study employs a special type of network called a *physics-informed neural PDE solver* [46, 55]. While neural solvers trained on labeled input-output pairs can learn the solution operator of a PDE, their ability to generalize to out-of-distribution data, such as boundary or initial conditions outside the training set, significantly increases the demand on the training dataset size. In SciML, data is often scarce and computationally expensive to generate. To address this challenge, physics-informed neural solvers adopt a similar strategy to PINNs by incorporating an additional PDE loss as a form of regularization. This *physics* loss effectively constrains the space of possible solutions, softly enforcing the PDE constraints. By incorporating domain knowledge into the training process, the model is more robust to noise and uncertainties present in the dataset. As an example, for the Laplace equation, the PDE loss for a batch of n collocation points $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\} \subseteq \Omega$ is defined as

$$
\mathcal{L}_{pde}(\boldsymbol{X}, \hat{\boldsymbol{g}}; \theta) = \frac{1}{n} \sum_{i}^{n} (\Delta \mathcal{N}(\boldsymbol{x}_i, \hat{\boldsymbol{g}}; \theta))^2 \tag{3}
$$

Intuitively, as the loss function is minimized during training, the PDE residual will approach zero, $\Delta \mathcal{N}(\boldsymbol{x}, \hat{\boldsymbol{g}}; \theta) \rightarrow 0$, indicating that the network approximately satisfies the Laplace equation $\Delta \mathcal{N}(\boldsymbol{x}, \hat{\boldsymbol{g}}; \theta) \approx \Delta u(\boldsymbol{x}) = 0$.

2.3 Domain Decomposition

Domain decomposition methods are widely used in solving boundary value problems [1, 9, 19]. These methods partition the domain of a BVP into smaller subdomains, and then iteratively combine solutions of the subdomains to develop the global solution. Domain decomposition methods enable scaling across multiple nodes, making them a powerful tool for scaling PDE solvers.

The classic example of domain decomposition is the alternating Schwarz method (ASM) [16, 47]. ASM relies on overlapping subdomains to ensure continuity and information propagation between subdomains. While Schwarz methods are commonly used as preconditioners for Krylov methods [6], in this work we use a variant of ASM as the solver.

Continuing with the Laplace example, the domain Ω can be partitioned into two subdomains Ω_1 and Ω_2, such that $\Omega_1 \cap \Omega_2 \neq \emptyset$. The subdomain interfaces are $\Gamma_1 = \partial\Omega_1 \cap \Omega_2$ and $\Gamma_2 = \Omega_1 \cap \partial\Omega_2$. To solve the global domain using ASM, the following routine is applied iteratively:

$$
\begin{aligned}
\Delta u_1^{n+1} &= 0 \text{ in } \Omega_1 \qquad & \Delta u_2^{n+1} &= 0 \text{ in } \Omega_2 \\
u_1^{n+1} &= u_2^n \text{ on } \Gamma_1 \qquad & u_2^{n+1} &= u_1^{n+1} \text{ on } \Gamma_2
\end{aligned} \tag{4}
$$

The superscript denotes the iteration of the solution: u_i^n is the solution of subdomain i at iteration n. The process alternates between solving for u_1 and u_2. The solution for u_1 is used to set the interface condition on Γ_2. Then, with this condition, we solve for u_2. Subsequently, the solution for u_2 is used to set the interface condition on Γ_1. Schwarz proved the convergence of this iterative scheme for general elliptic PDEs [47]. Lions proved that ASM can be used to solve systems with an arbitrary number of subdomains, and a parallel version of ASM exhibits similar convergence properties to the original Schwarz method [36]. However, it is worth noting that the convergence rate of Schwarz methods is signifiantly influenced by the mesh parameter and overlap. A system consisting of many subdomains with little overlap require more iterations to converge compared to a system with fewer subdomains and greater overlap. To address these issues, several improvements to the alternating Schwarz method have been proposed [11, 15]. We leave exploring these improvements to future work.

2.4 Mosaic Flow

Mosaic Flow [53] is a novel approach for solving BVPs on diverse domains with arbitrary boundary conditions. It consists of two primary components:

(1) The subdomain solver (**SDNet**) is a *physics-informed neural PDE solver* trained to solve a BVP on a small domain with arbitrary boundary conditions. Although this paper focuses on Dirichlet boundary conditions, SDNet can also be used with Neumann or Robin boundary conditions. SDNet's effectiveness arises from its ability to rapidly generate predictions for any point within the domain. Note that while the boundary function input to SDNet is discretized, the xy-coordinate can be in continuous space. This is unlike finite difference or finite elements methods, which require discretizing the interior of the domain [30].
(2) The *Mosaic Flow predictor* (**MFP**) illustrated in Figure 2 is an iterative algorithm that leverages pre-trained SDNet's inferences

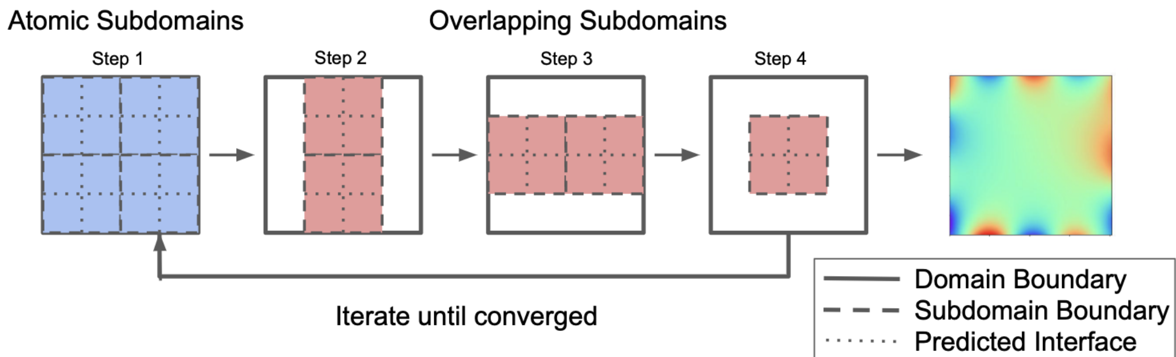


Figure 2: MFP predicts the solution in new domains by combining SDNet predictions on *atomic* and *overlapping* subdomains. Unlike traditional numerical methods and PINNs, MF predictor only infers the solution on interfaces of subdomains rather than computing solutions for all grid or collocation points in the domain during each iteration.

to solve BVPs on large domains—much larger than those that can be solved with a SDNet directly. The iterative algorithm decomposes the domain into smaller *atomic subdomains*, and updates the solution within each subdomain using SDNet's predictions. It also ensures the spatial regularity of the solution along the subdomain boundaries using *overlapping subdomains*, inspired by the alternating Schwarz method. By utilizing SDNet as the sub-domain solver, MFP inherits its ability to make predictions for arbitrary points within the domain. This feature results in a significant performance advantage, as Mosaic Flow can compute the solution for only a small fraction of grid points, specifically the interfaces of the subdomains, as opposed to all grid points in the entire domain, as done in classical ASM.

Mosaic Flow combines the efficiency of SDNet on small domains with the scalability of MFP on larger domains, enabling the efficient solution of complex BVPs.

3 NEURAL PDE SOLVER TRAINING

SDNet is a neural PDE solver designed to approximate solutions to boundary value problems by taking a discretized boundary condition $\hat{g}$ and the coordinates of a point $\boldsymbol{x}$ as inputs. $\mathcal{N}(\boldsymbol{x}, \hat{g}; \theta) \approx u(\boldsymbol{x}; g)$, where $u(\cdot; g)$ is the solution to the BVP determined by the boundary function g. By including the boundary condition in the input, the network can be used across multiple unseen instances of a BVP. However, this also results in a large input layer that, in combination with a PINN loss function, can make the network computationally expensive to train.

In the case of Mosaic Flow, the training of physics-informed neural PDE solvers is restricted to a small domain for each PDE type. However, even on small domains (e.g., spatial dimensions of 1×1), the training can take several hours (see Fig 6). To enable scalable training, performance-focused optimization and parallelization across multiple GPUs are crucial. By optimizing the training process, it becomes feasible to create a library of pre-trained SDNet models for different PDE types, facilitating the solution of complex multiphysics problems efficiently.

3.1 SDNet Model Overview

In general, our approach is agnostic of the choice of model for SDNet. For instance, one could use pure MLPs, a flavor of DeepONet [38], or Fourier layers [35]. The architecture of the neural solver used in this work, shown in Figure 3, is a variant of DeepONet that we inherit and improve. We first apply 1D convolutions to the input boundary conditions to create a high-dimensional embedding. The reason for using 1D convolutions is to take advantage of the inherent spatial structure of the boundary conditions, which can be seen as a 1D curve along the boundary of the domain. By convolving this signal, we capture local patterns and relevant features for predicting the solution within the domain. We anticipate this treatment of the boundaries will enhance convergence performance without affecting per-iteration performance, as convolutions are computationally efficient. We choose not to use Fourier layers due to the non-periodic nature of Dirichlet boundary conditions [35]. Although FNO can handle non-periodic boundaries, the combination of convolutions and fully-connected layers proves sufficient for capturing global phenomena.

Next, we apply the *input-split* optimization discussed in Section 3.2 to the high-dimensional boundary embedding coupled with the input $\boldsymbol{x}$. The rest of the architecture is composed of a stack of linear layers, each followed by a nonlinear activation function. We use the GELU activation function [20] because PINN training tends to have better convergence properties when using a smooth activation function [48].

3.2 Optimized Input Embedding

A common input embedding in physics-informed neural PDE solvers is to concatenate the spatial coordinates $\boldsymbol{x}$ with the discretized boundary function $\hat{g}$ into a single input vector [38, 53]. However, this *input-concat* approach is highly inefficient. For example, in a square 2D domain discretized into an $N \times N$ resolution, inferring the solution, $u(\boldsymbol{x})$ at a single point $\boldsymbol{x}$ in the domain requires an input vector of dimension $4N + 2$. The discretized boundary function $\hat{g}$ is a vector of dimension $4N$ and the additional 2 dimensions are for the xy-coordinates.

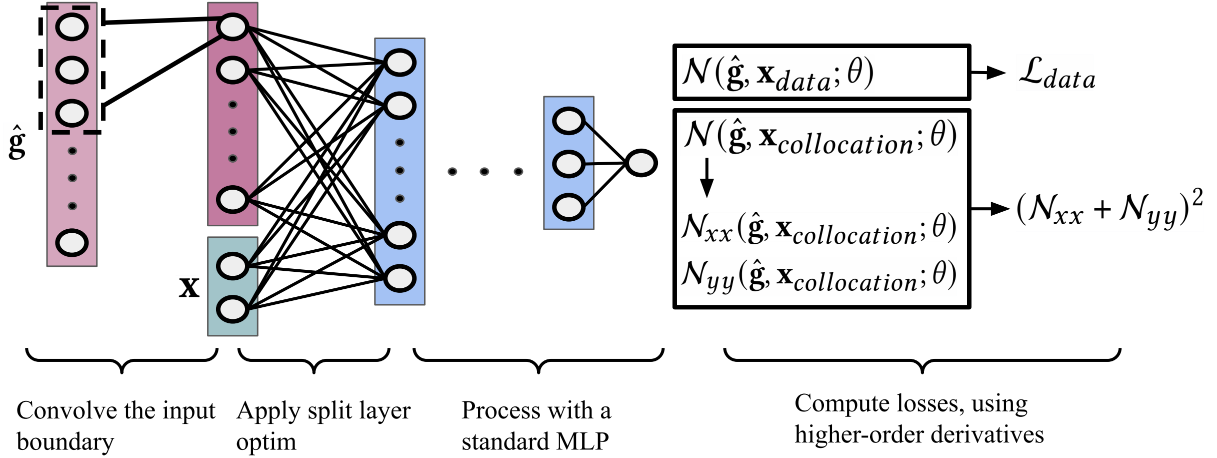


Figure 3: The architecture of the neural solver begins by convolving the input boundary condition, which results in a high-dimensional embedding. Next, the split layer optimization is applied to the batches of boundary conditions and xy-coordinates. The processed output is fed into a standard MLP, which approximates the solution of the PDE at the input points. Finally, the model computes higher-order derivatives to enforce the PDE constraints in the loss function.

When inferring the solution of q points in a batch, the input becomes a $q \times (4N + 2)$ matrix $\mathbf{I}$:

$$\mathbf{I} = \begin{bmatrix} \hat{\boldsymbol{g}}, \boldsymbol{x}_1 \\ \hat{\boldsymbol{g}}, \boldsymbol{x}_2 \\ \vdots \\ \hat{\boldsymbol{g}}, \boldsymbol{x}_q \end{bmatrix} = \begin{bmatrix} \mathrm{G} & \mathrm{X} \end{bmatrix} \tag{5}$$

where the matrix $\mathbf{G} \in \mathbb{R}^{q \times 4N}$ is formed by replicating the vector $\hat{\boldsymbol{g}}$ for each point in the batch and the rows of $\mathbf{X} \in \mathbb{R}^{q \times 2}$ are the coordinates of the q points. Denoting the weights of the first layer of the neural network as $\mathbf{W} \in \mathbb{R}^{d \times (4N+2)}$, the output of this layer is given by the matrix multiplication of $\mathbf{I}$ with $\mathbf{W}$, followed by the application of the activation function ϕ. Mathematically, we can express this as:

$$\mathbf{U} = \phi(\mathbf{I}\mathbf{W}^{\top}) \in \mathbb{R}^{q \times d} \tag{6}$$

To reduce the computational cost and remove the redundancy in $\mathbf{G}$ introduced by the *input-concat* approach, we split weight matrix $\mathbf{W} \in \mathbb{R}^{d \times (4N+2)}$ into two column blocks, denoted as $\mathbf{W}_1 \in \mathbb{R}^{d \times 4N}$ and $\mathbf{W}_2 \in \mathbb{R}^{d \times 2}$. Using eq. (5), we can rewrite eq. (6) to arrive at our optimized approach:

$$\mathbf{U} = \phi(\begin{bmatrix} \mathrm{G} & \mathrm{X} \end{bmatrix} \begin{bmatrix} \mathbf{W}_1^{\top} \\ \mathbf{W}_2^{\top} \end{bmatrix}) \tag{7}$$

$$= \phi(\hat{\boldsymbol{g}}\mathbf{W}_1^{\top} \oplus \mathbf{X}\mathbf{W}_2^{\top}) \tag{8}$$

where $\oplus$ is a broadcasted sum along the second axis of $\boldsymbol{X}\boldsymbol{W}_2^T$. Note that the discretized boundary condition $\hat{\boldsymbol{g}}$ is no longer replicated for each point in the input, but is computed only once and broadcasted along the batch dimension. This reduces the overall number of computations required by the network. With eq. (6), the cost of the first layer is $O(qNd)$. In comparison, with our optimized *input-split* approach in eq. (8), the cost is reduced to $O(Nd + qd)$. More importantly, this reduces the memory requirement of input tensor from $q(4N+2)$ words to $4N+2q$ words; when q and N are large, this saving can be substantial. The reduction in memory usage achieved by the optimized approach makes it possible to scale training to larger batch sizes.

3.3 Distributed Data Parallel Neural PDE solvers

After optimizing the network architecture, we accelerate the training of neural PDE solvers with a physics-informed loss using data parallelism. Recall that when training a physics-informed model, we use a loss function with multiple terms: $\mathcal{L}(\theta) = \mathcal{L}_{data}(\theta) + \mathcal{L}_{pde}(\theta)$ where $\mathcal{L}_{data}(\theta)$ represents the *data loss* function. The loss function that enforces the PDE constraints, $\mathcal{L}_{pde}(\theta)$ requires the computation of higher-order derivatives with respect to the model inputs. In the case of the Laplace equation, this involves calculating the second derivatives $\mathcal{N}_{xx}$ and $\mathcal{N}_{yy}$. This results in a large autograd graph that consumes significant device memory. The size of this autograd graph limits the batch size on a single GPU, motivating the need to scale up to multiple GPUs to improve performance. It is worth noting that without the PDE loss, a purely data-driven model could be trained with a larger batch size on a single device. However, such a model may exhibit physical inaccuracies and require significantly larger dataset for training.

To efficiently train the network with multiple loss terms, we separate the data and collocation points into distinct forward passes. This approach simplifies the application of different losses to their respective coordinates, as the data loss can only be applied to points with known solutions. However, when using distributed data parallelism (DDP), it is important to preserve the standard semantics of stochastic gradient descent (SGD). In data-parallel training, the model is replicated across different compute nodes, and local gradients are computed on each process. To synchronize gradients, an *allreduce* [4] operation is commonly used, where the gradients are averaged across processes. To maintain the correct semantics of SGD, we must be mindful of when gradient synchronization occurs. If synchronization occurs after both forward passes, it will compute a sum of averages rather than a true global average. Although this

approach may yield satisfactory results in practice, it does not offer the same convergence guarantees.

Algorithm 1 SDNet Training Iteration

1: $\hat{G}$ is a batch of discretized boundary functions
2: X_{data} is a batch of points with known solutions
3: $X_{collocation}$ is a batch of points with unknown solutions
4: **Step 1:** Solve Data Points
5: $P_{data} = \mathcal{N}(\hat{G}, X_{data}; \theta)$
6: $\nabla_{data} = \nabla\mathcal{L}_{data}(P_{data}; \theta)$
7: **Step 2:** Solve Collocation Points
8: $P_{collocation} = \mathcal{N}(\hat{G}, X_{collocation}; \theta)$
9: $\nabla = \nabla_{data} + \nabla\mathcal{L}_{pde}(P_{collocation}; \theta)$
10: **Step 3:** Perform *allreduce* on gradient ∇

To maintain consistency with SGD and ensure reliable convergence, we propose the method outlined in Algorithm 1 for each training iteration. In step 1, the forward and backward passes are computed for the data points (lines 5-6) on each process without averaging gradients across processes. Then, in step 2, we apply the forward and backward passes for the collocation points (lines 8-9). The gradients for the collocation points are accumulated onto the gradients for the data points (line 9), and this sum is subsequently averaged across all processes using the allreduce operation in step 3. The averaged gradients are applied locally on each device, ensuring consistency. The proposed approach not only ensures accurate gradient accumulation but also offers the advantage of performing a single allreduce operation per training iteration, instead of two separate operations for data and collocation points. This optimization reduces communication overhead and enhances the scalability and efficiency of the training process.

4 PARALLEL AND DISTRIBUTED INFERENCE

The baseline MFP [53] has limited scalability, as we show in Section 5.3. This constraint significantly hampers its capacity to solve BVPs on large domains. Our approach to addressing this limitation includes two strategies: increasing device-level parallelism and formulating the *distributed MF predictor* algorithm for multi-GPU scaling. The algorithm is designed to harness the inherent strengths of the baseline MFP, while simultaneously extending its scope to BVPs on significantly larger domains.

4.1 Batched Inference with Atomic Subdomains

The baseline MFP adopts a sequential approach to solve subdomains, which ensures that all predictions are based on updated boundary conditions. Empirical evidence suggests that relaxing this can have a negative effect on prediction accuracy. However, by observing Figure 2, it becomes evident that atomic subdomains within each iteration of the algorithm do not overlap. This creates an opportunity for concurrently predicting these non-overlapping subdomains. To leverage this, we implement a batching technique that combines the atomic subdomains as input for SDNet inference. This effectively increases GPU occupancy by exploiting device-level parallelism as demonstrated in Section 5.3.

4.2 Domain Parallelization for Distributed Inference

The MFP takes the boundary conditions of subdomains as its input. During the development of the distributed algorithm, a key factor is both accurately and effectively managing updates to boundary conditions within overlapping regions. The boundary information of the subdomains can be organized into a Cartesian grid. In the example illustrated in Figure 2, the distance between neighboring grid points is $\frac{1}{2}m$, which is a tunable hyperparameter. Choosing a smaller distance allows for more subdomains to be placed in the domain, potentially resulting in more accurate results. However, this also leads to increased computation and communication costs, as shown in Section 4.3. In this study, we choose a value of $\frac{1}{2}m$ because it is the largest distance we can use without significantly sacrificing accuracy.

To design a parallel algorithm for the MFP, we first divide the global domain into a 2D grid, where each block is assigned to a processor. The processors are assigned to this 2D grid in a row-wise scan pattern. It is worth noting that a processor mapping strategy based on locality-preserving space-filling curves such as Morton order [41] or Z-order could provide better load balancing and reduced data movement [45], although we leave this for future studies. We refer to the region owned by a processor as the *processor subdomain*. In order to iteratively approach the final solution using Schwarz methods, neighboring processor subdomains need to communicate boundary information. To enable this exchange of information, we give each processor additional *halo* boundary information from its neighboring processor subdomains. Figure 4 illustrates the distribution of processor subdomains and how the boundary information is exchanged between processors.

Our proposed distributed algorithm is outlined in algorithm 2, where t, ϵ, $\boldsymbol{g}$, SDNet, and $\boldsymbol{n}$ respectively denote the maximum number of iterations, convergence threshold, global boundary condition, the pre-trained SDNet model, and the neighbors of the current processor. We use the hat notation to denote a local variable (for example, $\hat{\boldsymbol{g}}_0$ denotes the local part of $\boldsymbol{g}_0$). The algorithm can be conceptually divided into two phases. The first phase is the iteration loop from line 2 to line 9. In each iteration, the boundary information of the local atomic subdomains is input to the pre-trained SDNet, which predicts the values *only* along the center lines of these atomic subdomains (line 3). Since the center lines of one subdomain are the boundary of another, these predictions are subsequently input to SDNet for the next iteration. After the inference step, the boundary information in the region where processor subdomains overlaps is packed into a contiguous buffer and sent to the corresponding neighbors with `communicate_new_boundaries` in line 4. Figure 4 provides a graphical illustration of which parts of the boundaries are being communicated. The second phase starts after t iterations or upon reaching the convergence threshold. In this phase, each processor uses the most recent atomic subdomain boundaries as input for SDNet, to predict the values at every grid point within each atomic subdomain, forming the local solution $\hat{\mathrm{S}}$ (line 10). Finally, an `all_gather` is performed to collect the distributed subdomains. In the region where processor subdomains overlap, the final solution is obtained by computing the average of the predictions.

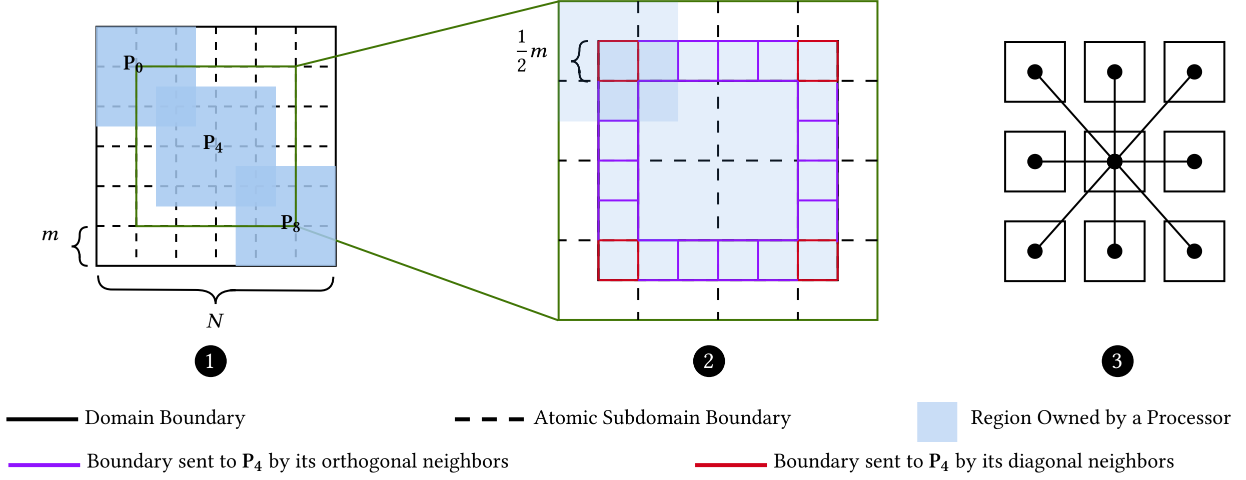


Figure 4: ❶ illustrates a $N \times N$ domain distributed among a 3×3 processor grid and the placement of non-overlapping atomic subdomains on the entire domain. We omitted some of the processors in these figures to avoid cluttering. ❷ zooms in on the region encompassing the subdomain owned by P_4. It highlights the boundaries sent to P_4 by its neighbors. The red lines indicate boundaries received by P_4 from its diagonal neighbors: $[P_0, P_2, P_6, P_8]$. The purple lines indicate boundaries received by P_4 from its orthogonal neighbors: $[P_1, P_3, P_5, P_7]$. ❸ shows the stencil communication pattern for exchanging boundary information. For processors on the four boundaries, the communication group will not include all 9 processors.

Algorithm 2 Distributed Mosaic Flow Predictor

```
1:  function PREDICT(t, ε, SDNet, g_0, n)
2:      for i in 1:t do
3:          ĝ_i = SDNet(ĝ_{i-1})
4:          ĝ_i = communicate_new_boundaries(ĝ_i, n)
5:          δ = ||ĝ_i − ĝ_{i-1}|| / ||ĝ_{i-1}||
6:          if δ < ε then
7:              break
8:          end if
9:      end for
10:     Ŝ = SDNet(ĝ_i)
11:     all_gather(Ŝ)
12:     S = combine all Ŝ and average over the overlapping regions
13:     return S
14: end function
```

To design a scalable algorithm, we partially relax the order of subdomain updates in the baseline algorithm to balance accuracy requirements with communication efficiency. In the algorithm illustrated in Figure 2, as subdomains are solved by SDNet, the update to the boundary information inside the subdomain is applied immediately. However, when processors solve subdomains on the overlapped region, the update to the boundary information cannot be reflected in the neighboring processor until the communication step. In our parallel algorithm, we relax this principle by communicating only once per iteration. This relaxation in synchronization not only reduces the communication frequency but also makes the communication pattern agnostic to subdomain placement schemes. It is worth highlighting that the baseline principle of immediate updates to boundary information still holds within individual processor subdomains. This relaxation is similar to the algorithm proposed by Lions [36], which was proven to converge to the global solution. Empirical results in Section 5.3 show that this modification does not prevent the distributed MFP from finding accurate solutions.

4.3 Cost Analysis

We now analyze the costs associated with the distributed MF Predictor. Suppose the global domain has a resolution of $N \times N$ and is distributed across P processors arranged in a $\sqrt{P} \times \sqrt{P}$ grid. The resolution of each subdomain is $m \times m$. As a result, each processor is assigned a subdomain consisting of $\frac{N}{m\sqrt{P}} \times \frac{N}{m\sqrt{P}}$ non-overlapping subdomains. Assuming that the subdomain boundaries form a Cartesian grid with interval $\frac{m}{d}$, and the overlapping region is $\frac{2(d-1)}{d}m$ wide along each subdomain boundary, there are $\frac{(dN)^2}{m^2P}$ subdomains in each processor with all eight neighbors. Using the alpha-beta model and removing the trailing terms, the communication cost of each processor is $C_{comm} = 8I\alpha + \frac{1}{\beta}I(16\frac{Nd}{\sqrt{P}})$, where α models the network latency and β models the network bandwidth.

Since communication is performed in every iteration, both the bandwidth and latency cost scale linearly with the iteration count. As the communication is limited to each processor and its immediate neighbors, the latency cost is not influenced by P or N. Bandwidth cost increases linearly with $\frac{N}{\sqrt{P}}$, which is the length of one side of each subdomain, and d, which controls how dense the subdomains are placed. Denoting the computation cost of making 1 SDNet inference as c, we can express the computation cost of each processor as $C_{comp} = c\frac{(dN)^2}{m^2P}$. From this, we expect the computation cost to scale linearly with the number of processors. Note that we assume communication can be carried out simultaneously with all neighbors in our analysis, which may not always hold in practice.

5 RESULTS AND DISCUSSION

We perform two sets of experiments to assess the performance of training and inference. First, we evaluate SDNet training across multiple GPUs to analyze per-iteration performance and the impact of scaling on convergence. We present results on multiple GPU clusters, as detailed in Table 2, to gain a deeper understanding of the impact of optimizations discussed in Section 3. Second, we evaluate the performance and scalability of distributed MFP on unseen domains that are significantly larger than the input seen by SDNet during training. Ground truth data for both experiments is generated using the approach described in Section 5.1.

	V100	A30	A100
Architecture	Volta	Ampere	Ampere
Peak FP32	14 TF	10.3 TF	19.5 TF
GPUs/node	4	4	2
Nodes	13	14	4
Memory	16GB (HBM2)	24GB (HBM2)	80GB (HBM2e)
Memory Bandwidth	900 GB/s	933 GB/s	2 TB/s
Intra-node Interconnect	32 GB/s (PCIe)	200 GB/s (NVLink)	600 GB/s (NVLink)
Inter-node Interconnect	100 Gbits/s (ConnectX-5 Infiniband)		

Table 2: GPU evaluation platforms and their specifications.

5.1 Data Generation

We generate two distinct datasets: one for training SDNet and another for evaluating the MFP. The training dataset consists of small domains of fixed size, while the test dataset includes larger domains of arbitrary sizes. To construct these datasets, we generate boundary conditions using Gaussian processes and follow a similar approach to the original Mosaic Flow paper [53]. First, we use a Sobol Sequence [50] to sample the hyperparameters of an infinitely differentiable Gaussian kernel of a 1-dimensional Gaussian process. Then, from each Gaussian process, we draw a sample function (i.e., a 1-D curve). This function serves as the discretized boundary function $\hat{\boldsymbol{g}}$ described in Section 2. Each boundary value problem for the Laplace equation is solved using pyAMG. [2].

5.2 SDNet Training

Train Dataset. We use the methodology described in Section 5.1 to generate a dataset of $20,000$ boundary conditions for domains with a resolution of 32×32 and spatial dimension of 0.5×0.5. The pairs of boundary conditions and sample solutions form our training dataset. We use 90% of this dataset for training and hold out the remaining 10% as a validation set.

Hyperparameters. We perform hyperparameter tuning to determine the optimal values for several parameters, including the maximum learning rate, the fraction of iterations used for learning rate warmup, the learning rate schedule, the number of epochs, weight decay, and the number of points per subdomain. We do this tuning using a single GPU and select a sufficiently large batch size to ensure efficient GPU utilization. The tuned hyperparameters we use are as follows: a maximum learning rate of 0.001, using 0.1% of iterations for learning rate warmup, using polynomial learning rate decay with the exponent set to one, training for 500 epochs, and setting the coefficient for weight decay to zero.

For experiments with varying GPU counts, we reuse the same hyperparameters from the single GPU case, with two modifications: (a) We scale the maximum learning rate by the square root of the increase in batch size. (b) The fraction of iterations used for learning rate warmup is scaled linearly with the increase in batch size [14, 57].

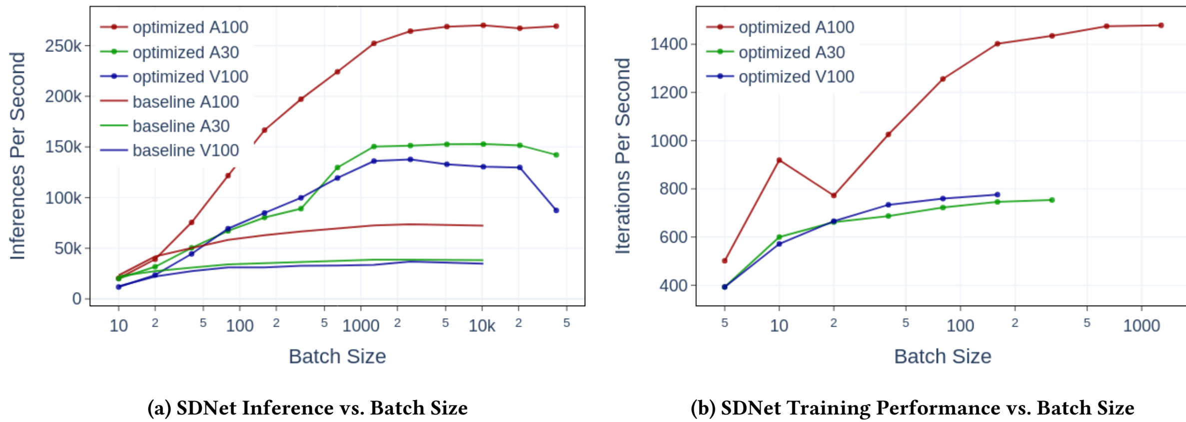
Finally, as we increase the number of GPUs, the number of points per batch can reach tens of thousands. We adopt the Lamb optimizer [57], which we find yields better convergence than AdamW [37] when scaling to larger batch sizes and multiple GPUs. Specifically, we utilize the implementation of FusedLAMB from Nvidia Apex.

Training Methodology. To train the SDNet, we employ a loss function with two terms: a data loss and a PDE loss. The data loss is a mean squared error using the pyAMG solution as the ground truth. The PDE loss is the PDE residual applied at the collocation points. It requires computing higher order derivatives with respect to the model inputs. Despite the relatively small size of our models compared to state-of-the-art vision and language models, the autograd graph generated during training consumes a significant amount of device memory. This memory constraint severely limits the batch size that can be used on a single GPU, which motivates the distributed data parallel approach to training. As seen in Figure 5, we can scale inference to process hundreds of thousands of subdomains at a time, but merely hundreds during training. Even for a relatively simple PDE like Laplace, a single model update requires three backward passes: (a) a backward pass to compute the derivatives w.r.t x and y, (b) a second backward pass to compute the *second* derivatives w.r.t. x and y and (c) a final backward pass, through the prior two gradient computations. We measure the maximum memory allocated during the forward and backward passes of the model. The results, presented in Table 3, highlight the difference in memory usage with and without the PDE loss. The inclusion of the PDE loss leads to a significant increase in memory consumption, primarily attributed to the storage of intermediate activations in the autograd graph.

# Domains	No PDE Loss	With PDE Loss
5	0.05 GB	0.503 G
320	2.77 GB	15.11 GB
640	5.54 GB	OOM

Table 3: Memory allocated during the forward pass, loss computation, and backward pass on a single V100 GPU with and without PDE loss. OOM indicates "out of memory".

We implement data parallel training using PyTorch Distributed. A key advantage of PyTorch's implementation of DDP training is the ability to overlap communication with the current backward pass [33]. This is unlike other frameworks, like Horovod, which overlap communication with the following forward pass. It is important to ensure that communication overhead does not dominate the overall execution time. Since our models are relatively small,



(a) SDNet Inference vs. Batch Size

(b) SDNet Training Performance vs. Batch Size

Figure 5: SDNet inference and training performance with varying batch sizes. *optimized* model utilizes the split-layer optimization, while the *baseline* model is a standard neural PDE solver. Each point is the average of 30 trials. The variance is near zero in every case. This plot shows both how the split-layer optimization improves performance, and enables scaling to larger batch sizes. For instance, the baseline models reach memory limits at a batch size of $10,000$ points, while the optimized models can scale to larger batch sizes, processing up to $50,000$ points during inference.

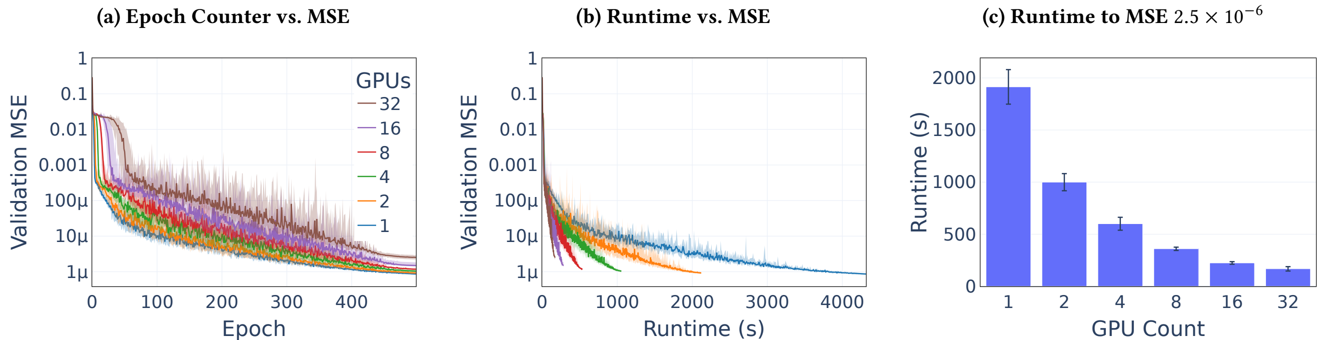


Figure 6: SDNet multi-GPU performance and impact on convergence. (a) shows the MSE of the validation set as a function of the epoch count. (b) illustrates the runtime improvements with increasing A30 GPUs. Both (a) and (b) report the median validation MSE across 10 models initialized with different random seeds. The bands represent the 95% confidence interval of the median [22, 31]. Note that (a) and (b) are plotted on a log_{10} scale and all models achieve final MSEs within 1.5×10^{-6} of the single GPU case. (c) shows the average time, across 10 trials, taken by each model to reach an MSE of 2.5×10^{-6}, which corresponds to the mean MSE of the final epoch with 32 A30 GPUs. The bands in (c) represent the standard deviation.

the forward passes are typically inexpensive. Therefore, overlapping communication with the current backward pass improves the efficiency of training our models.

Training Performance. We implemented several optimizations that result in much faster training compared to a baseline neural solver. First, we implemented the split layer, which significantly reduces redundant computation in the first layer of the network. This optimization is also important for the performance of model inference in the MFP, as seen in Figure 5. Second, we apply a series of 1-dimensional convolutions to the input boundary conditions, which form a smooth curve. Convolutions are cheap to compute, so this optimization has essentially no effect on the per-iteration performance of the MFP, but improves the convergence rate of the SDNet. Finally, we scale model training across multiple GPUs.

Figure 6 shows the performance and accuracy of SDNet when scaling the number of GPUs. Although, we observe a slight negative impact on the validation MSE, all models achieve final MSEs within 1.5×10^{-6} of the model trained on a single A30 GPU. Notably, the model trained on one GPU takes over 30 minutes to reach an MSE of 2.5×10^{-6}, while 32 GPUs reduces the training time to just *two minutes* to reach the same MSE, resulting in a speedup of 12×.

To compare the effectiveness of the SDNet models as sub-domain solvers for MFP, we additionally evaluate each SDNet on test problems of different sizes, as shown in Figure 7. Despite the slight variations in the validation set's MSE (see Figure 6), we observe consistent MAE across all models. This indicates that all models exhibit comparable accuracy and are equally reliable as sub-domain solvers for MFP.

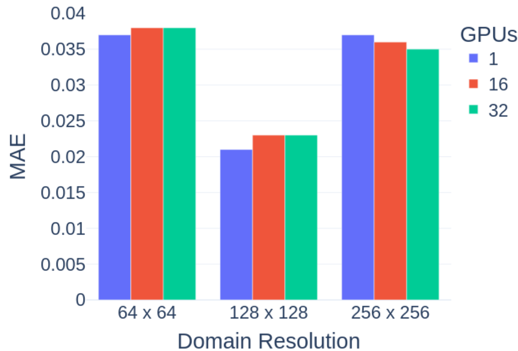


Figure 7: MAE of the MFP, using models trained with varying GPU count. The discretized boundary function for each domain is $\hat{g}(x) = sin(2\pi x)$. This illustrates that the small changes in MSE seen in Figure 6 have little affect on the MFP, which makes prediction of similar quality with each model.

5.3 MF Predictor Performance

We implement the distributed MFP in Python. For GPU-to-GPU communication, we use `mpi4py` [5], which is built with a CUDA-aware MPI library to enhance communication performance. To generate boundary conditions and ground truth solutions of the Laplace equation on larger domains, we use the method described in Section 5.1. We evaluate both batched inference for device-level parallelism and distributed inference for node-level parallelism.

Batched Inference. In this experiment, we assess the performance improvement achieved by batching the atomic subdomains during each iteration of the MFP (as discussed in Section 5.3). We compare this *batched* approach to the original *unbatched* algorithm, which predicts one subdomain at a time using SDNet. The results in Figure 8, shows the impact of batching when scaling the domain size from 1×2 to 16×16 (i.e., resolutions from 64×128 to 1024×1024). In the unbatched approach, time increases linearly with the domain size. However, with batching subdomains, we observe a significant improvement in GPU utilization, resulting in about 50% of the peak performance. Note that since atomic subdomain inferences are independent, batching improves performance by up to $100\times$ without sacrificing accuracy.

Distributed Inference. We conduct both strong and weak scaling studies to evaluate MFP on multiple GPUs. In the strong scaling experiments, we consider a BVP for the Laplace equation with a spatial domain size of 32×32 (2048×2048 resolution). This domain is divided into 4096 atomic subdomains where each subdomain is of size 0.5×0.5. The global boundary condition is generated using the same process described in Section 5.1. The MFP terminates when

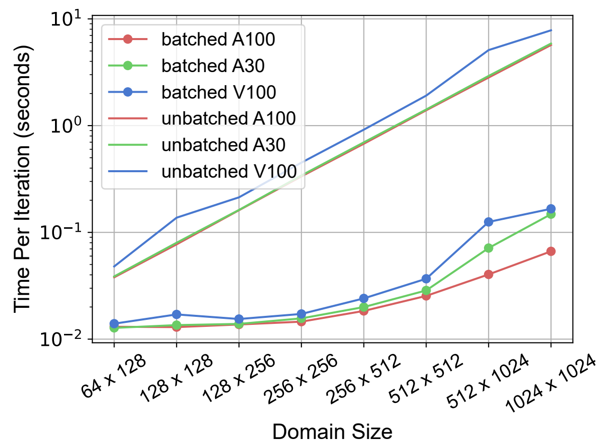


Figure 8: Performance of batched vs. unbatched atomic subdomains on a single GPU with increasing domain sizes. Time per iteration is calculated by averaging over 100 iterations.

the MAE of the solution drops below 0.05. The results, shown in Figure 9a, demonstrate a clear trend of decreasing computation time and an increasing percentage of communication time as we scale from 1 to 32 GPUs. The total runtime reduces from approximately 15 minutes (~ 880 seconds) to less than 2 minutes (~ 90 seconds), resulting in a speedup of almost $10\times$ on 32 A30 GPUs.

As discussed in Section 4.2, updates in the overlapping regions along the borders of processor subdomains are not immediately reflected since the data is distributed. Therefore, as we decompose a domain into more (and smaller) processor subdomains, a larger percentage of the boundary information becomes stale. This can lead to a degradation of the convergence rate of the distributed MFP algorithm. In the strong scaling experiment, we investigate the impact of the distributed algorithm on the convergence rate. We record the number of iterations required to reach an MAE of 0.05 and present the results in Table 4. As the number of processors increases, we observe a slight increase in the number of iterations required to reach the specified MAE. However, note that the benefits of parallelization and the reduction in computation time outweigh the slight increase in the number of iterations, leading to improved overall performance.

GPU Count	1	2	4	8	16	32
Iterations	3200	3250	3250	3300	3400	3500

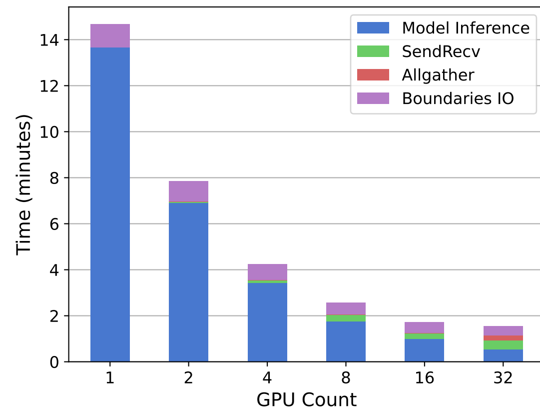
Table 4: The number of iterations required to achieve a MAE of 0.05 for different GPU counts. The corresponding runtimes are shown in Figure 9a.

We also perform a weak scaling experiment with an increasing number of processors while keeping the spatial size of each processor subdomain fixed at 16×8 (1024×512 resolution), this result is shown in Figure 9b. Computation scales well, as the only additional computation cost is to average across regions where processor domains overlap. However, the communication scaling is less optimal. We see around $4\times$ increase going from 2 to 8 GPUs, which then plateaus. This increase is likely due to high latency cost as the number of messages sent by each processor increases with

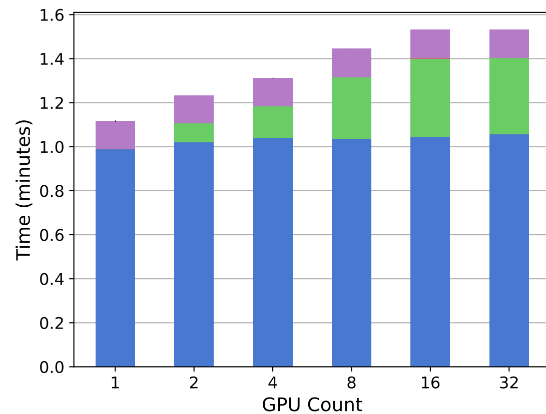
an increasing number of neighbors from 2 to 8 GPUs. We don't see a noticeable improvement in performance with CUDA-aware MPI compared to standard MPI, potentially due to the small buffer sizes of `send/recv` communication where latency dominates the overall communication performance [5]. The increased latency cost is further exacerbated by `mpi4py`, which serializes Pytorch tensors before communication. Techniques that leverage direct GPU-to-GPU communication through NVSHMEM [23] are potential alternatives to reduce this communication overhead.

Open problems. *Systems challenges* – One approach to addressing the latency overhead is to convert a latency-bound algorithm to a bandwidth-bound algorithm. This can be achieved by reducing the communication frequency. In the current implementation, each processor exchanges boundary information with its neighbors during every iteration. However, communicating less frequently introduces a trade-off with redundant computation. Given that compute scales significantly better than communication (both bandwidth and latency), *communication-avoiding algorithms* are worth exploring. Nonetheless, there is a communication lower bound that cannot be avoided, in which case, overlapping communication with computation can further push the scaling ceiling. It is worth noting that *communication-overlapping algorithms* have been well-studied in the context of numerical simulations [23, 45, 52]. However, neural PDE solvers can be significantly faster than numerical solvers. In contrast to large language models, current neural models for approximating PDEs are notably smaller. Additionally, batched inference only requires a forward pass (no expensive higher-order gradient computation). Consequently, communication becomes the *bottleneck* for scaling even on smaller GPU clusters. Studying the trade-offs of communication-avoiding and communication-overlapping algorithms in the context of distributed neural PDE solvers remains a promising direction for future research.

Algorithmic challenges – For BVPs, where you are interested in finding a solution that satisfies specific boundary conditions within a domain, information needs to be exchanged across the entire domain. For this reason, one-level Schwarz methods require a global coarse grid correction to scale to a large number of subdomains for solving BVPs [10]. FBPINN extended to multiple levels of overlapping domain decomposition demonstrates improved accuracy, specifically for large number of subdomains, implying that coarse levels are necessary for efficient global information propagation in large domains [8]. However, for time-dependent problems, where the solution evolves over time, information typically only needs to be exchanged between neighboring subdomains. As time advances, information is propagated across the domain as adjacent subdomains continually share their updated information. We hypothesize that distributed Mosaic Flow coupled with one-level Schwarz is optimal for exploring neural domain decomposition methods to solve time-dependent PDEs [18, 51].



(a) Strong scaling over a 2048×2048 resolution domain.



(b) Weak scaling for 2000 iterations where each GPU owns a 1024×512 resolution subdomain.

Figure 9: Strong and weak scaling of MFP. We average the results across 5 trials. "Boundaries IO" refers to reading subdomain boundaries for SDNet and updating them with the prediction from SDNet.

6 CONCLUSIONS

The hybrid parallelization scheme presented in this paper shows promise in scaling physics-informed neural PDE solvers to large domains using a combination of data parallel training and domain parallelization. The SDNets can be trained in minutes, allowing for the creation of a library of models for different PDEs. The MF Predictor demonstrated accuracy when scaling up to 32 GPUs. Overall, this work opens up avenues for future research in the field of physics-informed machine learning. There is still room for improvement by exploring other domain decomposition methods and improved Schwarz methods, such as using a coarse correction [10] or Optimized Schwarz methods [15], and extending DDM for time-dependent neural PDE solvers.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under the award number 2211908. We gratefully acknowledge the GPU computing resources provided on HPC3, a high-performance computing cluster operated by the Research Cyberinfrastructure Center at the University of California, Irvine. We specifically thank Hengjie Wang at Modular for helpful discussions on Mosaic Flow.

REFERENCES

[1] Satish Balay, Shrirang Abhyankar, Mark Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, W Gropp, et al. 2019. PETSc users manual. (2019).

[2] Nathan Bell, Luke N. Olson, and Jacob Schroder. 2022. PyAMG: Algebraic Multigrid Solvers in Python. *Journal of Open Source Software* 7, 72 (2022), 4142. https://doi.org/10.21105/joss.04142

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

[4] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Van De Geijn. 2007. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience* 19, 13 (2007), 1749–1783.

[5] Lisandro Dalcin and Yao-Lung L Fang. 2021. mpi4py: Status update after 12 years of development. *Computing in Science & Engineering* 23, 4 (2021), 47–54.

[6] Victorita Dolean, Martin J Gander, Walid Kheriji, Felix Kwok, and Roland Masson. 2016. Nonlinear preconditioning: How to use a nonlinear Schwarz method to precondition Newton's method. *SIAM Journal on Scientific Computing* 38, 6 (2016), A3357–A3380.

[7] Victorita Dolean, Alexander Heinlein, Siddhartha Mishra, and Ben Moseley. 2022. Finite basis physics-informed neural networks as a Schwarz domain decomposition method. *arXiv preprint arXiv:2211.05560* (2022).

[8] Victorita Dolean, Alexander Heinlein, Siddhartha Mishra, and Ben Moseley. 2023. Multilevel domain decomposition-based architectures for physics-informed neural networks. *arXiv preprint arXiv:2306.05486* (2023).

[9] Victorita Dolean, Pierre Jolivet, and Frédéric Nataf. 2015. *An introduction to domain decomposition methods: algorithms, theory, and parallel implementation.* SIAM.

[10] Olivier Dubois, Martin Gander, Sébastien Loisel, Amik St-Cyr, and Daniel Szyld. 2009. The Optimized Schwarz Method with a Coarse Grid Correction. *SIAM Journal on Scientific Computing* 34 (11 2009). https://doi.org/10.1137/090774434

[11] Olivier Dubois, Martin J Gander, Sébastien Loisel, Amik St-Cyr, and Daniel B Szyld. 2012. The optimized Schwarz method with a coarse grid correction. *SIAM Journal on Scientific Computing* 34, 1 (2012), A421–A458.

[12] Vikas Dwivedi, Nishant Parashar, and Balaji Srinivasan. 2021. Distributed learning machines for solving forward and inverse problems in partial differential equations. *Neurocomputing* 420 (2021), 299–316.

[13] Lawrence C. Evans. 2010. *Partial differential equations.* American Mathematical Society, Providence, R.I.

[14] Steven Farrell, Murali Emani, Jacob Balma, Lukas Drescher, Aleksandr Drozd, Andreas Fink, Geoffrey Fox, David Kanter, Thorsten Kurth, Peter Mattson, Dawei Mu, Amit Ruhela, Kento Sato, Koichi Shirahata, Tsuguchika Tabaru, Aristeidis Tsaris, Jan Balewski, Ben Cumming, Takumi Danjo, Jens Domke, Takaaki Fukai, Naoto Fukumoto, Tatsuya Fukushi, Balazs Gerofi, Takumi Honda, Toshiyuki Imamura, Akihiko Kasagi, Kentaro Kawakami, Shuhei Kudo, Akiyoshi Kuroda, Maxime Martinasso, Satoshi Matsuoka, Henrique Mendonça, Kazuki Minami, Prabhat Ram, Takashi Sawada, Mallikarjun Shankar, Tom St. John, Akihiro Tabuchi, Venkatram Vishwanath, Mohamed Wahib, Masafumi Yamazaki, and Junqi Yin. 2021. MLPerf HPC: A Holistic Benchmark Suite for Scientific Machine Learning on HPC Systems.. In *IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC).* 1–45.

[15] Martin J Gander. 2006. Optimized schwarz methods. *SIAM J. Numer. Anal.* 44, 2 (2006), 699–731.

[16] Martin Jakob Gander. 2008. Schwarz methods over the course of time. *Electronic transactions on numerical analysis* 31 (2008), 228–255.

[17] Ehsan Haghighat and Ruben Juanes. 2021. SciANN: A Keras/TensorFlow wrapper for scientific computations and physics-informed deep learning using artificial neural networks. *Computer Methods in Applied Mechanics and Engineering* 373 (2021), 113552.

[18] Sheikh Md Shakeel Hassan, Arthur Feeney, Akash Dhruv, Jihoon Kim, Youngjoon Suh, Jaiyoung Ryu, Yoonjin Won, and Aparna Chandramowlishwaran. 2023. BubbleML: A Multi-Physics Dataset and Benchmarks for Machine Learning. *arXiv preprint arXiv:2307.14623* (2023).

[19] Frédéric Hecht. 2012. New development in FreeFem++. *Journal of numerical mathematics* 20, 3-4 (2012), 251–266.

[20] Dan Hendrycks and Kevin Gimpel. 2016. Gaussian Error Linear Units (GELUs). *arXiv preprint arXiv:1606.08415* (2016).

[21] Oliver Hennigh, Susheela Narasimhan, Mohammad Amin Nabian, Akshay Subramaniam, Kaustubh Tangsali, Zhiwei Fang, Max Rietmann, Wonmin Byeon, and Sanjay Choudhry. 2021. NVIDIA SimNet™: An AI-accelerated multi-physics simulation framework. In *Computational Science–ICCS 2021: 21st International Conference, Krakow, Poland, June 16–18, 2021, Proceedings, Part V.* Springer, 447–461.

[22] Torsten Hoefler and Roberto Belli. 2015. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15).* Association for Computing Machinery, New York, NY, USA, Article 73, 12 pages. https://doi.org/10.1145/2807591.2807644

[23] Ismayil Ismayilov, Javid Baydamirli, Doğan Sağbili, Mohamed Wahib, and Didem Unat. 2023. Multi-GPU Communication Schemes for Iterative Solvers: When CPUs Are Not in Charge. In *Proceedings of the 37th International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23).* Association for Computing Machinery, New York, NY, USA, 192–202. https://doi.org/10.1145/3577193.3593713

[24] Ameya D Jagtap and George E Karniadakis. 2021. Extended Physics-informed Neural Networks (XPINNs): A Generalized Space-Time Domain Decomposition based Deep Learning Framework for Nonlinear Partial Differential Equations.. In *AAAI Spring Symposium: MLPS.* 2002–2041.

[25] Ameya D Jagtap, Ehsan Kharazmi, and George Em Karniadakis. 2020. Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems. *Computer Methods in Applied Mechanics and Engineering* 365 (2020), 113028.

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

[27] Ehsan Kharazmi, Zhongqiang Zhang, and George Em Karniadakis. 2021. hp-VPINNs: Variational physics-informed neural networks with domain decomposition. *Computer Methods in Applied Mechanics and Engineering* 374 (2021), 113547.

[28] Nikola B. Kovachki, Zongyi Li, Burigede Liu, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Andrew M. Stuart, and Anima Anandkumar. 2021. Neural Operator: Learning Maps Between Function Spaces. *CoRR* abs/2108.08481 (2021).

[29] Aditi Krishnapriyan, Amir Gholami, Shandian Zhe, Robert Kirby, and Michael W Mahoney. 2021. Characterizing possible failure modes in physics-informed neural networks. *Advances in Neural Information Processing Systems* 34 (2021), 26548–26560.

[30] Stig Larrson and Vidar Thomée. 2003. *Partial Differential Equations with Numerical Methods.* Springer Berlin, Heidelberg.

[31] Jean-Yves Le Boudec. 2010. *Performance Evaluation of Computer and Communication Systems.* EPFL Press, Lausanne, Switzerland. https://doi.org/10.1201/b16328

[32] Ke Li, Kejun Tang, Tianfan Wu, and Qifeng Liao. 2019. D3M: A deep domain decomposition method for partial differential equations. *IEEE Access* 8 (2019), 5283–5294.

[33] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch distributed: experiences on accelerating data parallel training.. In *Proceedings of the VLDB Endowment.* 3005–3018.

[34] Wuyang Li, Xueshuang Xiang, and Yingxiang Xu. 2020. Deep domain decomposition method: Elliptic problems. In *Mathematical and Scientific Machine Learning.* PMLR, 269–286.

[35] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. 2020. Fourier Neural Operator for Parametric Partial Differential Equations. *ICLR.*

[36] Pierre-Louis Lions et al. 1988. On the Schwarz alternating method. I. In *First international symposium on domain decomposition methods for partial differential equations*, Vol. 1. Paris, France, 42.

[37] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations.* https://openreview.net/forum?id=Bkg6RiCqY7

[38] Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George Em Karniadakis. 2021. Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature machine intelligence* 3, 3 (2021), 218–229.

[39] Lu Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis. 2021. DeepXDE: A deep learning library for solving differential equations. *SIAM review* 63, 1 (2021), 208–228.

[40] Stefano Markidis. 2021. The old and the new: Can physics-informed deep-learning replace traditional linear solvers? *Frontiers in big Data* (2021), 92.

[41] G. M. Morton. 1966. A Computer Oriented Geodetic DataBase and a New Technique in File Sequencing. *Tech.rep.,IBM,* (1966). https://dominoweb.draco.res.ibm.com/0dabf9473b9c86d48525779800566a39.html

[42] Ben Moseley, Andrew Markham, and Tarje Nissen-Meyer. 2021. Finite Basis Physics-Informed Neural Networks (FBPINNs): a scalable domain decomposition approach for solving differential equations. *arXiv preprint arXiv:2107.07871* (2021).

[43] Octavi Obiols-Sales, Abhinav Vishnu, Nicholas Malaya, and Aparna Chandramowliswharan. 2020. CFDNet: A deep learning-based accelerator for fluid simulations. In *Proceedings of the 34th ACM international conference on supercomputing.* 1–12.

[44] Octavi Obiols-Sales, Abhinav Vishnu, Nicholas P Malaya, and Aparna Chandramowlishwaran. 2021. SURFNet: Super-resolution of turbulent flows with transfer learning using small datasets. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT).* IEEE, 331–344.

[45] Johannes Pekkilä, Miikka S Väisälä, Maarit J Käpylä, Matthias Rheinhardt, and Oskar Lappi. 2022. Scalable communication for high-order stencil computations using CUDA-aware MPI. *Parallel Comput.* 111 (2022), 102904.

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

[47] Hermann Amandus Schwarz. 1869. Ueber einige Abbildungsaufgaben. (1869).

[48] Yeonjong Shin, Jermone Darbon, and George Karniadakis. 2020. On the convergence of physics informed neural networks for linear second-order elliptic and parabolic type PDEs.. In *Communications in Computational Physics*. 2042–2074.

[49] Khemraj Shukla, Ameya D Jagtap, and George Em Karniadakis. 2021. Parallel physics-informed neural networks via domain decomposition. *J. Comput. Phys.* 447 (2021), 110683.

[50] I.M. Sobol. 1998. On quasi-Monte Carlo integrations. *Mathematics and Computers in Simulation* 47, 2 (1998), 103–112. https://doi.org/10.1016/S0378-4754(98)00096-2

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

[52] Hengjie Wang and Aparna Chandramowlishwaran. 2020. Pencil: A pipelined algorithm for distributed stencils. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[53] Hengjie Wang, Robert Planas, Aparna Chandramowlishwaran, and Ramin Bostanabad. 2022. Mosaic flows: A transferable deep learning framework for solving PDEs on unseen domains. *Computer Methods in Applied Mechanics and Engineering* 389 (2022), 114424.

[54] Sifan Wang, Yujun Teng, and Paris Perdikaris. 2021. Understanding and mitigating gradient flow pathologies in physics-informed neural networks. *SIAM Journal on Scientific Computing* 43, 5 (2021), A3055–A3081.

[55] Sifan Wang, Hanwen Wang, and Paris Perdikaris. 2021. Learning the solution operator of parametric partial differential equations with physics-informed DeepONets. *Science Advances* 7, 40 (2021). https://doi.org/10.1126/sciadv.abi8605

[56] Sifan Wang, Xinling Yu, and Paris Perdikaris. 2022. When and why PINNs fail to train: A neural tangent kernel perspective. *J. Comput. Phys.* 449 (2022), 110768.

[57] Yang You, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. 2020. Large Batch Optimization for Deep Learning: Training BERT in 76 minutes.. In *ICLR*. 2042–2074.

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT IDENTIFICATION

Mosaic Flows is a novel domain decomposition method for scaling physics-informed operator networks to large domains. It contains two parts: a Subdomain Network (SDNet) and the Mosaic Flows (MF) predictor. SDNet is an operator network trained to solve a certain partial differential equation on a small domain. The MF predictor decomposes a large domain into smaller overlapping subdomains and uses SDNet to solve these subdomains. After many iterations of solving overlapping subdomains, it will converge to the PDE solution. In this work, we propose methods to substantially accelerate both the training of SDNet and MF predictor through distributed computing and sequential optimizations.

The source code in our artifact implements the methods described in the article and we provide all of our visualization scripts. We provide training scripts and sample hyperparameters that can be used to reproduce the results for training performance scaling and accuracy. Similarly, we provide a script implementing the MF Predictor. Pre-trained checkpoints are provided, which will reproduce the reported accuracies.

The SDNet and the MF predictor are implemented with Python and rely heavily on PyTorch and its new distributed communication library. We use PyTorch's automatic differentiation to compute the higher-order derivatives. For each training run we write errors and iteration runtimes to a TensorBoard log file. From this, we reconstruct per-iteration latencies and times for a full training run.

The performance results reported in the article are also generated directly through testing the software described here. Using the appropriate parameters, which are provided in sample scripts, the accuracy of the SDNet can be reproduced approximately. Similarly, the SDNet checkpoints provided can be used to reproduce the accuracy of MF predictor. While they can not be reproduced exactly, the general performance trends for both SDNet and MF predictor should also be reproducible on any hardware.

Generally, the artifacts we provide will enable reproducing the trends of our experiments, but will generally not allow for exact reproduction.

REPRODUCIBILITY OF EXPERIMENTS

There are two separate workflows in our work: one is for training the SubDomain Network (SDNet) and the second is for testing the Mosaic Flows (MF) Predictor.

Subdomain Network (SDNet) Training. First, we describe the steps for training. Our training scripts require a Linux cluster with at least interconnected 32 GPUs, Python, NCCL, and an assortment of common Python Libraries (such as PyTorch). When using Nvidia V100 or A30 GPUs, The training time can vary from several minutes to several hours depending on the number of GPUs used and the target accuracy. To reproduce the training experiments, one must run the submission script with the desired number of GPUs. The training script saves checkpoints for the model every 200 epochs. Additionally, it checkpoints the model that achieves the lowest validation error and the final model state after the last iteration.

All of the runtime data will be logged automatically. The TensorBoard log files include run times for each training iteration and various error metrics like the mean squared error, mean absolute error, and PDE residuals. The visualization scripts take the corresponding log directory as a command-line argument. The visualization scripts parse the TensorBoard log files and can construct the error plots with respect to the number of iterations, wall time, or number of epochs.

The results produced by the visualization scripts should look quite similar to the plots presented in the paper. Some figures are the combination of several separate plots, but each individual plot should match nicely with the reproduction. The main training script is "gfnet_torch_distributed.py" and can be run on a slurm cluster using the sample submissions script "torch_dist_loss_limit.sh".

Mosaic Flows (MF) Predictor. The following steps describes how to reproduce the strong scaling result presented in the article. This workflow requires a Linux cluster with at least 32 GPUs with python, jupyter, NCCL, and the required python libraries such as PyTorch properly installed. Each of the 3 experiments mentioned below typically takes less than an hour to finish, however this time may vary depending on the software and hardware configuration of the system used.

(1) In a cluster with the Slurm job scheduling system, navigate to `src/mf` inside the code repository.
(2) Run `sbatch strongScaling.sub` to submit a job that runs the MF predictor to solve the Laplace equation on the 32×32 domain with increasing number of GPUs. (This job script is written for Slurm, if the system uses other job scheduling tools, changes to the job script have to be made before running it.)
(3) The output of this job will be two series of `.csv` files in the `src/mf/figure` directory. The first series contains the time performance of each run. The second contains the accuracy of each run.
(4) Run the jupyter script `sc23-mosaic-flows/src/mf/figure/strongScaling.ipynb` to reproduce the figures.

In general, the reproduced result should show the runtime decreases linearly with the number of processors used. Running the steps above should reproduce the exact same accuracy result presented in the article.

Weak scaling can be reproduced with very similar steps mentioned above with the following changes:

(1) Run `sbatch weakScaling.sub` to submit the job.
(2) Run `sc23-mosaic-flows/src/mf/figure/weakScaling.ipynb` to reproduce the figure.

In this case, the runtime of each run is expected to increase slightly as the number of processors increase, with most of the increase coming from the increase in communication cost.

The experiment results that compares the sequential optimization can also be reproduced with very similar steps with the following changes:

(1) Run `sbatch batchVunBatch.sub` to submit the job.
(2) Run `sc23-mosaic-flows/src/mf/figure/batchVsUnbatched.ipynb` to reproduce the figure.

In this case, we can expect to see the runtime of the batched version to be significantly shorter than the runtime of the unbatched version.