ADCME MPI: Distributed Machine Learning for Computational Engineering

Kailai Xu,[1] Eric Darve[1, 2]
{kailaix, darve}@stanford.edu
[1]Institute for Computational and Mathematical Engineering, Stanford University, Stanford, CA 94305, USA;
[2] Mechanical Engineering, Stanford University, Stanford, CA 94305, USA

Abstract

We propose a framework for training deep neural networks (DNNs) that are coupled with partial differential equations (PDEs) in a parallel computing environment. Unlike most distributed computing frameworks for DNNs, our focus is to parallelize both numerical solvers and DNNs in forward and adjoint computations. Our parallel computing model views data communication as a node in the computational graph for numerical simulations. The advantage of our model is that data communication and computing are cleanly separated, which enables better flexibility, modularity, and testability of the software. We demonstrate our approach on a large-scale problem and show that we can achieve substantial acceleration by using parallel numerical PDE solvers while training DNNs that are coupled with PDEs.

Introduction

Deep neural networks (DNNs) have been demonstrated to be very effective for solving inverse problems in computational engineering (Raissi, Perdikaris, and Karniadakis 2019; Meng et al. 2020; Pakravan et al. 2020). In our previous work (Xu, Huang, and Darve 2020; Xu and Darve 2019; Huang et al. 2020; Fan et al. 2020), we successfully combined numerical solvers and deep neural networks for data-driven inverse modeling. Mathematically, we consider an implicit model $F(f, u) = 0$ where f is an unknown function, which we approximate using a DNN. We denote the DNN $\mathcal{N}_\theta$, where θ is the neural network weights and biases; u is a function which depends on f through $F(f, u) = 0$. We are given some (partial) observations u_{obs} of the function u, which are used to optimize θ and minimize the difference between f and $\mathcal{N}_\theta$. The optimization problem is formulated as a PDE-constrained optimization problem

$$\begin{aligned} &\min_\theta L(u) \quad (u \text{ is indirectly a function of } \theta) \\ &\text{such that } F(\mathcal{N}_\theta, u) = 0 \end{aligned} \tag{1}$$

$L(u)$ is a loss function, which measures the discrepancy between hypothetical and actual observations. For example, in this paper we use the square loss function $L(u) = \|u - u_{\text{obs}}\|_2^2$.

One advantage of such a formulation is that the known physics, such as the physical laws described by PDEs, are preserved to the largest extent and solved with well-developed and efficient numerical solvers. Meanwhile, we can leverage the approximation power of DNNs.

To solve this optimization problem, one can first solve the physics constraint $F(\mathcal{N}_\theta, u) = 0$ in Equation (1) numerically and then plug the solution $u(\theta)$ into $L(u)$. This leads to an unconstrained optimization problem

$$\min_\theta \tilde{L}(\theta) = L(u(\theta))$$

We developed a Julia (Bezanson et al. 2017) library, ADCME (Xu and Darve 2020a), with a TensorFlow (Abadi et al. 2016) automatic differentiation backend to solve problems of this type. ADCME expresses both the numerical solver and DNNs using computational graphs. Therefore, the gradient $\nabla_\theta \tilde{L}(\theta)$ can be calculated automatically by back-propagating gradients through both the numerical solvers[1] and DNNs. In this paper, we use "operator" and "node" interchangeably to refer to a node in the computational graph, which is a function that takes incoming edges (intermediate data) as inputs and outputs outgoing edges (intermediate data).

However, one challenge with this approach is that for large-scale problems, the memory and computational costs for the numerical solver are prohibitive. The de-facto standard for solving such large-scale problems on modern distributed memory high performance computing (HPC) architectures is the Message Passing Interface (MPI) (Gabriel et al. 2004; Gropp, Thakur, and Lusk 1999). The TensorFlow backend used by ADCME was originally designed for deep learning/machine learning. Despite that there are much work on extending TensorFlow for distributed training of machine learning models, some of the key capabilities, such as distributed linear algebra and domain decomposition, for solving scientific computing problems are still lacking. This paper is about incorporating MPI functionalities into ADCME to achieve scalability and flexibility for distributed memory.

[1]For details on how the gradient back-propagation works for numerical solvers in ADCME, we refer readers to (Xu and Darve 2020b).

The main idea is to parallelize numerical solvers by splitting the mesh or matrices onto different MPI ranks (or processors). Then, data communication nodes are inserted into the computational graph. Because for our computational engineering applications DNNs are typically small, they are duplicated on each processor. Each computational graph includes a set of "communication" nodes (Fig. 1-top), which are absent in a single processor computational graph. These operators invoke MPI calls and are in charge of data communication between different computer nodes. During the gradient back-propagation, we need to reverse the data-flow direction and operation of the data communication operators (Wang and Pothen 2015; Utke et al. 2009) (Fig. 1-bottom).

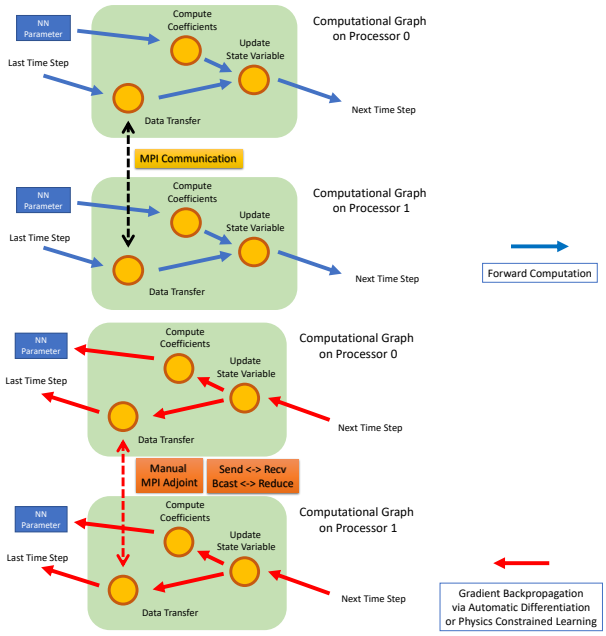

Figure 1: Paradigm for distributed machine learning for computational engineering. The data communication operations are treated as nodes in the computational graph. They are separate from the other computing operators. This makes it easy to reuse existing serial implementations of the numerical PDE solver.

Distributed Computing Models

There are many existing work and software for distributed computing with deep neural networks (Griebel and Zumbusch 1999; Notay 1995; Douglas, Haase, and Langer 2003) and numerical PDEs (Bekkerman, Bilenko, and Langford 2011; Jordan and Mitchell 2015). The two domains have quite different distributed computing models due to distinct features of targeted applications. ADCME MPI embraces a hybrid model that is suitable for inverse modeling in computational engineering because our method for solving Equation (1) requires a combination of the above two models.

In deep learning, one major challenge is that datasets are too large to fit into memory. Therefore, both datasets and computational loads are distributed onto different machines and mini-batch optimization algorithms, such as stochastic gradient descent (Bottou 2010), are used. Each processor calculates predictions and gradients, which are aggregated on one or more processors. To further scale out in a limited bandwidth environment, parameter servers (Li et al. 2013, 2014), where each server stores a part of the parameters, are implemented. There are also extensive work on model parallism, which parallelizes the computation by splitting the DNNs into multiple parts (Chen, Yang, and Cheng 2018; Hewett and Grady II 2020).

The parallel computing model in computational engineering is quite different from deep learning. The computational engineering applications feature data communication across neighboring points in a mesh (domain decomposition) or different parts of a matrix. In terms of computational graph, this pattern indicates that there are many more communications besides the reduction of gradients at the end. This motivates us to design new distributed computing models for computational engineering inverse modeling applications.

Methodology

ADCME MPI aims at providing a modular, efficient, and flexible implementation for distributed computing in inverse modeling. Conceptually, we can treat data communication operations as a node in the computational graph: they are similar to computational nodes (e.g., a linear solver), except that their responsibility is to invoke MPI calls and back-propagate the gradients in the reverse mode automatic differentiation (Baydin et al. 2017). This solution provides an elegant enhancement to the ADCME library because to convert a single processor program to multiple processor one, users only need to insert data communication nodes as needed and most parts of the original codes are unchanged. In this section, we briefly describe our contributions in ADCME MPI to extend its distributed computing capabilities to couple DNNs and PDE solvers.

MPI APIs

ADCME MPI provides a set of commonly used MPI primitives, such as `mpi_bcast`, `mpi_gather`, `mpi_send`, etc. These operators are wrappers for standard MPI APIs. However, these operators are also "differentiable," in the sense that they can handle gradient back-propagation. The gradient back-propagation functionality uses the fact that there exists one-to-one correspondence between forward and backward MPI calls. For example, the forward "send" corresponds to the backward "receive" (Cheng 2006; Towara, Schanen, and Naumann 2015).

Halo Exchange

To enable the communication between adjacent patches in a mesh, ADCME provides efficient implementations of data communication operators for halo exchange patterns (Fig. 2). Halo exchange patterns are very common in scientific computing because numerical solvers often involve communication between neighboring points (Bianco 2014). We use a nonblocking send/receive strategy. In the gradient back-propagation phase, this order of send/receive is reversed.

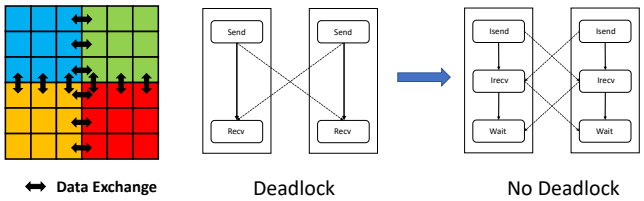

Figure 2: Left: domain decomposition and halo exchange pattern. Two adjacent patches exchange boundary information. Right: using nonblocking sends/receives to avoid deadlocks.

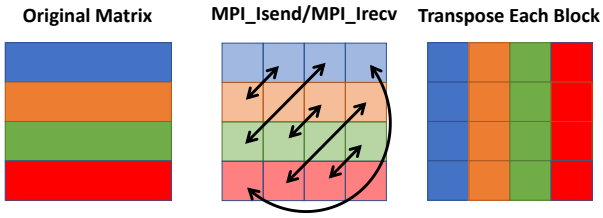

Figure 3: Transposition of a distributed sparse matrix. Each block exchanges nonzero entries with the corresponding transposed block. The original and resulting matrices are both stored in CSR formats.

Matrix Transposition

Sparse matrices are very important tools in computational engineering. ADCME MPI stores large sparse matrices as CSR matrices and each MPI processor stores a portion of rows with continuous row indices.

For reverse-mode automatic differentiation, matrix transposition is an operator that is common in gradient back-propagation. For example, assume the forward computation is ($\mathbf{x}$ is the input, $\mathbf{y}$ is the output, and $\mathbf{A}$ is a matrix)

$$\mathbf{y} = \mathbf{A}\mathbf{x} \tag{2}$$

Given a loss function $L(\mathbf{y})$, the gradient back-propagation calculates

$$\frac{\partial L(\mathbf{y}(\mathbf{x})))}{\partial \mathbf{x}} = \frac{\partial L(\mathbf{y})}{\partial \mathbf{y}}\frac{\partial \mathbf{y}(\mathbf{x})}{\partial \mathbf{x}} = \frac{\partial L(\mathbf{y})}{\partial \mathbf{y}}\mathbf{A}$$

Here $\frac{\partial L(\mathbf{y})}{\partial \mathbf{y}}$ is a row vector, and therefore

$$\left(\frac{\partial L(\mathbf{y}(\mathbf{x})))}{\partial \mathbf{x}}\right)^T = \mathbf{A}^T\left(\frac{\partial L(\mathbf{y})}{\partial \mathbf{y}}\right)^T$$

requires a matrix vector multiplication, where the matrix is $\mathbf{A}^T$.

The transposition of a distributed sparse matrix is implemented in three steps (Fig. 3):

1. The submatrix owned by each MPI processor is split into subblocks. Meta information (e.g., number of nonzeros in each block) is collected.
2. Each block B exchanges the meta information with the target block, where B^T should be placed.
3. Each subblock is transposed and the data are transferred to the target block.

Distributed Optimization

In general, the objective function of our problem can be written as a sum of local objective functions

$$\min_\theta \tilde{L}(\theta) = \sum_{i=1}^{N} f_i(\theta)$$

where f_i is the local objective function, N is the number of processors.

Despite many existing distributed optimization algorithm, in this work we adopt a simple approach: aggregating gradients $\nabla_\theta f_i(\theta)$ and updating θ on the root processors. Fig. 4 shows how we can convert an existing optimizer to an MPI-enabled optimizer. The basic idea is to let the root processor notify worker processors whether to compute the loss function or the gradient. Then the root processor and workers will collaborate on executing the same routines and thus ensuring the correctness of collective MPI calls.

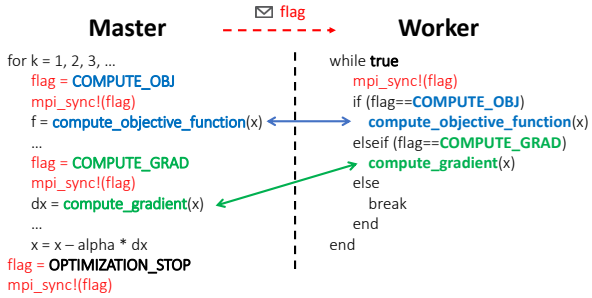

Figure 4: Refactoring a serial optimizer to an MPI-based optimizer in our framework.

Numerical Benchmarks

As a demonstration, we present a benchmark result with ADCME MPI. The example shows that the overhead introduced by ADCME is very small compared to the actual computation.

The governing equation is given by Poisson's equation

$$\begin{aligned} \nabla\cdot(\kappa(\mathbf{x})\nabla u(\mathbf{x})) &= f(\mathbf{x}) \qquad \mathbf{x}\in\Omega \\ u(\mathbf{x}) &= 0 \qquad \mathbf{x}\in\partial\Omega \end{aligned} \tag{3}$$

Here $f(\mathbf{x})\equiv 1$, and $\kappa(\mathbf{x})$ is approximated by a deep neural network

$$\kappa(\mathbf{x}) = \mathcal{N}_\theta(\mathbf{x})$$

where θ is the neural network weights and biases. The equation is discretized using the finite difference method on a uniform grid and the discretization leads to a linear system

$$\mathbf{A}(\theta)\,\mathbf{u} \approx \mathbf{f} \tag{4}$$

$\mathbf{u}$ is the solution vector and $\mathbf{f}$ is the source vector. Note that $\mathbf{A}$ is a sparse matrix and its entries depend on θ.

The sparse matrix is constructed using the differentiable halo exchange operator and stored as a distributed CSR matrix. Equation (4) is solved using the algebraic multigrid method in Hypre (Falgout and Yang 2002). During the gradient back-propagation, we need to solve a linear system with the coefficient matrix $\mathbf{A}^T$. This matrix is obtained using the technique described in the last section.

In the strong scaling experiments, we consider a fixed problem size 1,800 $\times$ 1,800 (mesh size, which implies the matrix size is around 32 million $\times$ 32 million). In the weak scaling experiments, each MPI processor owns a 300 $\times$ 300 block. For example, a problem with 3,600 processors has the problem size 90,000 $\times$ 3,600 $\approx$ 0.3 billion.

We first consider the weak scaling case. We consider two cases: each MPI rank has 1 core or 4 cores. In the latter case, the TensorFlow backend enjoys the benefit of inter-parallelism, where independent operators in the computational graph can be executed simultaneously. However, 4 cores do not guarantee a 4 times acceleration; the performance depends on the availability of independent tasks, scheduling conflicts, resource contention, etc. Fig. 5 shows the runtime for the forward computation as well as the gradient back-propagation. There are two important observations:

1. By using more cores per processor, the runtime is reduced significantly. For example, the runtime for the backward is reduced to around 10 seconds from 30 seconds by switching from 1 core to 4 cores per processor.

2. The runtime for the backward pass is typically less than twice the forward computation. Although the backward pass requires solving two linear systems (one of them is in the forward computation), the AMG (algebraic multigrid) linear solver in the back-propagation may converge faster, and therefore may cost less than during the forward pass.

Additionally, we show the overhead in Fig. 6, which is defined as the difference between total runtime and Hypre linear solver time, for both the forward and backward calculation.

We see that the overhead is quite small compared to the total time, especially when the problem size is large. This indicates that the ADCME MPI implementation is very effective.

In Fig. 7, we consider the strong scaling. In this case, we fixed the whole problem size and split the mesh onto different MPI processors. Fig. 7 shows the runtime for the forward computation and the gradient back-propagation. We can reduce the runtime by more than 20 times for the expensive gradient back-propagation by utilizing more than 100 MPI processors. Fig. 8 shows the speedup and efficiency. We can see that the 4 cores have smaller runtime compared to 1 core [2].

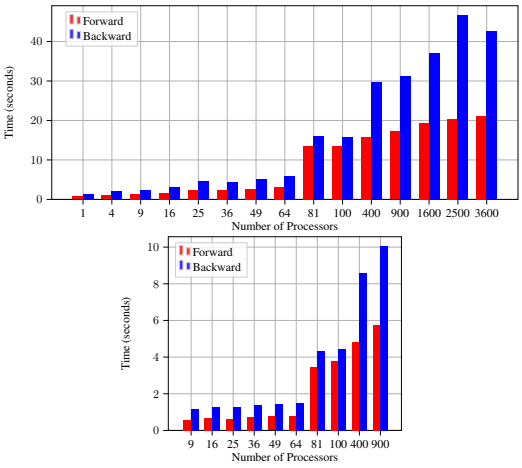

Figure 5: Weak scaling runtime for forward computation and gradient back-propagation. Top: each MPI processor has 1 core; bottom: each MPI processor has 4 cores. We see a jump near 64 cores because this is where network communications start to take place (recall each of our CPUs has 32 cores, and each node has two CPUs). The plots show results of weak scaling, each MPI processor solves a local problem of the same size.

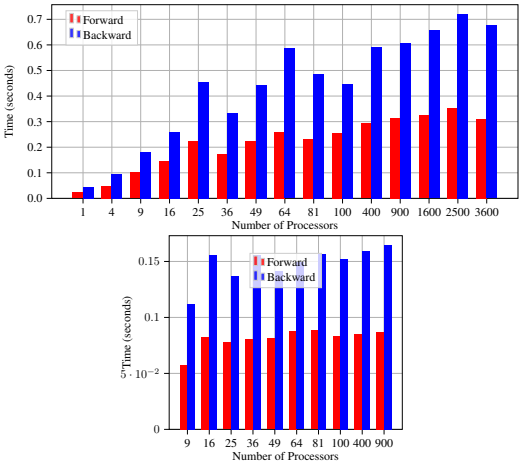

Figure 6: Overhead of ADCME for matrix solving in Hypre. The top and bottom bar plots correspond to the top and bottom ones in Fig. 5.

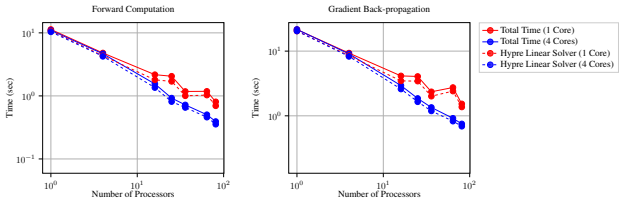

Figure 7: Runtime for forward computation and gradient back-propagation. The plots show results of strong scaling, the whole problem size is fixed and we increase the number of MPI processors (each processor has 1 or 4 cores).

[2]Finding a scaling sweet spot for a mixed programming model (MPI and OpenMP) of Hypre AMG solvers on multicore clusters is challenging (Baker, Schulz, and Yang 2010). The intra- and inter-parallelism of the TensorFlow backend also add difficulties to finding a scaling strategy.

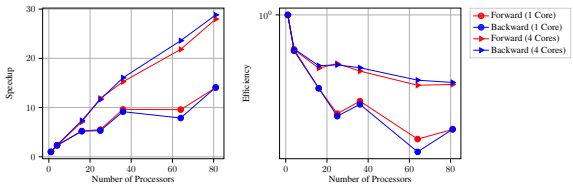

Figure 8: Speedup and efficiency for parallel computing of the Poisson's equation.

Conclusion

We presented the functionalities of ADCME MPI. Our benchmark results show that the overhead introduced by ADCME for distributed computing programs is very small compared with the computing time. The ADCME MPI distributed computing solution is quite flexible, allowing users to use custom parallel algorithms or libraries at their discretion. With the advent of experimental techniques that enable gathering large amounts of data, deep neural network based data-driven modeling will become essential tools for scientific discovery. The growing dataset and problem size add another level of challenges. Therefore, ongoing work on distributed computing in machine learning for computational engineering remains an important and promising direction in the foreseeable future.

Acknowledgements

This work is supported by the Applied Mathematics Program within the Department of Energy (DOE) Office of Advanced Scientific Computing Research (ASCR), through the Collaboratory on Mathematics and Physics-Informed Learning Machines for Multiscale and Multiphysics Problems Research Center (DE-SC0019453).

References

Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 265–283.

Baker, A. H.; Schulz, M.; and Yang, U. M. 2010. On the performance of an algebraic multigrid solver on multicore clusters. In *International Conference on High Performance Computing for Computational Science*, 102–115. Springer.

Baydin, A. G.; Pearlmutter, B. A.; Radul, A. A.; and Siskind, J. M. 2017. Automatic differentiation in machine learning: a survey. *The Journal of Machine Learning Research* 18(1): 5595–5637.

Bekkerman, R.; Bilenko, M.; and Langford, J. 2011. *Scaling up machine learning: Parallel and distributed approaches*. Cambridge University Press.

Bezanson, J.; Edelman, A.; Karpinski, S.; and Shah, V. B. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59(1): 65–98.

Bianco, M. 2014. An interface for halo exchange pattern. *www. prace-ri. eu/IMG/pdf/wp86. pdf* .

Bottou, L. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, 177–186. Springer.

Chen, C.-C.; Yang, C.-L.; and Cheng, H.-Y. 2018. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform. *arXiv preprint arXiv:1809.02839* .

Cheng, B. N. 2006. A duality between forward and adjoint MPI communication routines. .

Douglas, C. C.; Haase, G.; and Langer, U. 2003. *A tutorial on elliptic PDE solvers and their parallelization*. SIAM.

Falgout, R. D.; and Yang, U. M. 2002. hypre: A library of high performance preconditioners. In *International Conference on Computational Science*, 632–641. Springer.

Fan, T.; Xu, K.; Pathak, J.; and Darve, E. 2020. Solving Inverse Problems in Steady State Navier-Stokes Equations using Deep Neural Networks. *arXiv preprint arXiv:2008.13074* .

Gabriel, E.; Fagg, G. E.; Bosilca, G.; Angskun, T.; Dongarra, J. J.; Squyres, J. M.; Sahay, V.; Kambadur, P.; Barrett, B.; Lumsdaine, A.; et al. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, 97–104. Springer.

Griebel, M.; and Zumbusch, G. 1999. Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves. *Parallel Computing* 25(7): 827–843.

Gropp, W.; Thakur, R.; and Lusk, E. 1999. *Using MPI-2: advanced features of the message passing interface*. MIT press.

Hewett, R. J.; and Grady II, T. J. 2020. A Linear Algebraic Approach to Model Parallelism in Deep Learning. *arXiv preprint arXiv:2006.03108* .

Huang, D. Z.; Xu, K.; Farhat, C.; and Darve, E. 2020. Learning constitutive relations from indirect observations using deep neural networks. *Journal of Computational Physics* 109491.

Jordan, M. I.; and Mitchell, T. M. 2015. Machine learning: Trends, perspectives, and prospects. *Science* 349(6245): 255–260.

Li, M.; Andersen, D. G.; Park, J. W.; Smola, A. J.; Ahmed, A.; Josifovski, V.; Long, J.; Shekita, E. J.; and Su, B.-Y. 2014. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 583–598.

Li, M.; Zhou, L.; Yang, Z.; Li, A.; Xia, F.; Andersen, D. G.; and Smola, A. 2013. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, volume 6, 2.

Meng, X.; Li, Z.; Zhang, D.; and Karniadakis, G. E. 2020. PPINN: Parareal physics-informed neural network for time-dependent PDEs. *Computer Methods in Applied Mechanics and Engineering* 370: 113250.

Notay, Y. 1995. An efficient parallel discrete PDE solver. *Parallel computing* 21(11): 1725–1748.

Pakravan, S.; Mistani, P. A.; Aragon-Calvo, M. A.; and Gibou, F. 2020. Solving inverse-PDE problems with physics-aware neural networks. *arXiv preprint arXiv:2001.03608* .

Raissi, M.; Perdikaris, P.; and Karniadakis, G. E. 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: 686–707.

Towara, M.; Schanen, M.; and Naumann, U. 2015. MPI-Parallel Discrete Adjoint OpenFOAM. In *ICCS*, 19–28.

Utke, J.; Hascoet, L.; Heimbach, P.; Hill, C.; Hovland, P.; and Naumann, U. 2009. Toward adjoinable MPI. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, 1–8. IEEE.

Wang, M.; and Pothen, A. 2015. High Order Automatic Differentiation with MPI. *Dep* 3: v0.

Xu, K.; and Darve, E. 2019. The neural network approach to inverse problems in differential equations. *arXiv preprint arXiv:1901.07758* .

Xu, K.; and Darve, E. 2020a. ADCME: Learning Spatially-varying Physical Fields using Deep Neural Networks.

Xu, K.; and Darve, E. 2020b. Physics constrained learning for data-driven inverse modeling from sparse observations. *arXiv preprint arXiv:2002.10521* .

Xu, K.; Huang, D. Z.; and Darve, E. 2020. Learning constitutive relations using symmetric positive definite neural networks. *arXiv preprint arXiv:2004.00265* .