Applying Recent Machine Learning Approaches to Accelerate the Algebraic Multigrid Method for Fluid Simulations

Thorben Louw(✉) and Simon McIntosh-Smith(✉)

Department of Computer Science, University of Bristol, Bristol BS8 1UB, UK
{thorben.louw.2019,S.McIntosh-Smith}@bristol.ac.uk

Abstract. In this work, we describe our experiences trying to apply recent machine learning (ML) advances to the Algebraic Multigrid (AMG) method to predict better prolongation (interpolation) operators and accelerate solver convergence. Published work often reports results on small, unrepresentative problems, such as 1D equations or very small computational grids. To better understand the performance of these methods on more realistic data, we create a new, reusable dataset of large, sparse matrices by leveraging the recently published Thingi10K dataset of 3D geometries, along with the FTetWild mesher for creating computational meshes that are valid for use in finite element method (FEM) simulations. We run simple 3D Navier-Stokes simulations, and capture the sparse linear systems that arise.

We consider the integration of ML approaches with established tools and solvers that support distributed computation, such as HYPRE, but achieve little success. The only approach suitable for use with unstructured grid data involves inference against a multi-layer message-passing graph neural network, which is too memory-hungry for practical use, and we find existing frameworks to be unsuitable for efficient distributed inference. Furthermore, the model prediction times far exceed the complete solver time of traditional approaches. While our focus is on inference against trained models, we also note that retraining the proposed neural networks using our dataset remains intractable.

We conclude that these ML approaches are not yet ready for general use, and that much more research focus is required into how efficient distributed inference against such models can be incorporated into existing HPC workflows.

Keywords: HPC-AI · AMG · GNN

1 Introduction

In this paper, we describe our experience applying recently proposed machine learning (ML) models to predict better interpolation operators for the algebraic

This work was funded by the Engineering and Physical Sciences Research Council (EPSRC) via the Advanced Simulation and Modelling of Virtual Systems (ASiMoV) project, EP/S005072/1.

J. Nichols et al. (Eds.): SMC 2021, CCIS 1512, pp. 40–57, 2022.
https://doi.org/10.1007/978-3-030-96498-6_3

multigrid (AMG) method, with the aim of reducing the number of iterations the method takes to converge. Our interest lies in speeding up the solution of large, sparse linear systems which arise during fluid simulations.

A common criticism of recent ML-based approaches to solving partial differential equations (PDEs) or accelerate scientific simulations is that they are typically demonstrated on small 1D or 2D structured grids. Yet industrial and scientific simulations currently solve problems many orders of magnitude larger, and existing tools are designed to support *unstructured* 3D meshes. To be useful, ML acceleration approaches must be shown to work on these inputs. In this work, we use the 10,000 3D geometries in the Thingi10K dataset, run a simple 3D Navier-Stokes FEM fluid simulation on each, and capture the sparse matrices which result, to create a new dataset of sparse linear systems for evaluating such ML approaches.

In 2020, Luz et al. reported using a graph neural network (GNN) approach for accelerating AMG that is able to deal with unstructured data [24]. Since they found good generalization to domains outside of their original training regime, we use their implementation as a basis for our work.

Machine learning models such as the one in [24] learn good values for parameters in a process called *training*. Training is much more computationally demanding than *inference* against a trained model, so it attracts the bulk of research attention today. However, the researcher wanting to use or deploy a trained model for inference is left with a significant amount of software engineering effort to make their model to run well, and this is not often discussed.

A disconnect exists between the way high-performance computing (HPC) tools are designed to scale in over the distributed compute nodes in a supercomputer – typically using implementations of the Message Passing Interface (MPI) to communicate over fast interconnects and are invoked via workload managers such as Slurm – and the support for serving machine learning models in the dominant ML frameworks, which favor a cloud-like distributed computing setup. Model serving frameworks load trained, optimized models and make them available to clients through an interface such as gRPC. They optimize for latency and throughput, dynamically batching requests and scaling compute resources to match demand, and allow optimized models to make use of acceleration hardware, using hardware-specific frameworks such as NVIDIA's TensorRT that optimizes models to make use of TensorCores on NVIDIA accelerators. Alternatively, one might wish to simply load a model locally in an application by integrating modern machine learning frameworks directly in source code. Even so, the post-training optimization process can involve serializing to a model interchange format like ONNX [25], sparsification, data-type reduction and quantization, and running models through deep learning (DL) compilers such as TVM [5] to optimize computation and data movement. This difficulty is reflected in the growth of a software engineering trend called "MLOps" (Machine Learning Operations), which recognizes that the devising and training models is only a small part of the overall effort in making these models useful [22,26].

Our attempt at using the approach in [24] with our dataset fails: the message-passing graph neural network (GNN) model and the graph neural network framework used for the implementation do not scale to support the size of matrices in our dataset. An inference against the model – one small part of the problem setup phase – runs longer than a traditional solver takes to complete the whole solution, and uses huge amounts of memory even for modest problem sizes. In addition, the model does not generalize well to even the smaller sparse matrices in our dataset. We find that support for distributed exact inference using large graphs in GNN frameworks today is limited, and difficult to integrate into our workflow. The GNN nature of the model makes it difficult to optimize, and support in the common ONNX interchange format is lacking.

The structure of the rest of this paper is as follows: we give an overview of AMG in Sect. 2.1 and discuss our evaluation dataset in Sect. 2.2. We review of the recently proposed ML acceleration approaches for AMG in Sect. 3 with special focus on the approaches of Greenfeld et al. [16] and Luz et al. [24]. We discuss our results in Sect. 4 before concluding with some thoughts on future work.

Our contributions in this work are

- A new dataset of 30,000 sparse linear systems representative of the fluid mechanics simulations arising from FEM simulation complex 3D geometries. The dataset is easy to modify for different problems and boundary conditions, and useful for practically evaluating suggestions for ML-accelerated solvers.
- Findings from our experience with the model from Greenfeld et al. [16], which demonstrate that much simpler models can also learn good interpolation operators, with corresponding benefits for integrating them into applications
- A description of our experience trying to apply a Graph Neural Network such as the one in [24] to larger, 3D fluid dynamics problems.

2 Background

2.1 Overview of Algebraic Multigrid

In this section we give a very brief the AMG concepts which are important for contextualizing the ML methods which follow. A thorough overview of AMG is available in [29].

In this setting, we are interested in solving linear systems of the familiar form:

$$A\mathbf{x} = \mathbf{b} \tag{1}$$

where $A \in \mathbb{R}^{n \times n}$ is a very sparse matrix. These systems frequently arise in scientific domains when solving partial differential equations (PDEs) that are discretized on a computational grid, and then solved using techniques such as FEM, or the Finite Difference Method (FDM) or Finite Volume Method (FVM).

For example, in a typical fluid dynamics solver, such linear systems are solved at each timestep when calculating the next grid values for the velocity and pressure fields.

In real-world problems these linear systems are both very large (many millions of rows), and extremely sparse, encouraging the use of specialized solvers which take advantage of these properties. Iterative *relaxation* methods are used, which improve on an initial guess until the solution converges (i.e. until the residual $e_i = \|A\mathbf{x}_i - \mathbf{b}\|$ becomes sufficiently small).

Many relaxation methods have been developed over the last few decades, including the well-known class of Krylov subspace methods such as the Conjugate Gradient (CG) method and Generalized Minimal Residual Method (GMRES).

Relaxation methods are known to be good at reducing high-frequency errors (associated with eigenvectors of A that have large eigenvalues), but poor at reducing the smooth or low-frequency errors, and can need many iterations to converge.

Multigrid methods improve on this situation by creating a hierarchy of coarser (smaller) grid levels. At each level, residual errors after applying smoothing at the fine grid level are projected to the next coarsest grid using a problem- and level-specific 'restriction operator' R, before applying smoothing (relaxation) at the coarser grid level. This process is repeated, resulting in smaller and smaller linear systems, until eventually a system is small enough to solve directly. The coarsest grid error is then interpolated back from coarse to fine grid levels using problem- and level-specific 'prolongation operators' P and added back to the post-smoothing guess as a correction, until finally a few relaxations are applied to the finest system. This entire process forms one "V-cycle", several iterations of which are performed until the residual converges.

Coarse-grid correction smooths out the low-frequency errors, while relaxations at the finest grid level smooth out the high-frequency errors, resulting in rapid convergence.

In the original Geometric Multigrid context, the problem geometry ('grid') is available to the solver, but the Algebraic Multigrid method [28] extends this multilevel idea to general linear systems in a 'black-box' fashion, using only the entries of the matrix A and the target vector $\mathbf{b}$ as inputs. Coarsening strategies in AMG use algebraic properties of the nodes in the matrix (such as the strength-of-connection or energy-based properties) rather than the problem geometry to select coarse nodes, and heuristic methods to select the weights in the prolongation operator.

AMG's black-box approach may be less efficient than approaches which can exploit the problem geometry, but it has an important software engineering advantage: easily re-usable, optimized libraries for the solution of general linear systems from different domains can be created which use standard matrix-vector interfaces. Examples of such libraries in the HPC world are HYPRE's BoomerAMG solver [11] and ML [13], which have developed parallel versions of AMG algorithms that support distributed sparse matrices for very large systems, and scale well over thousands of nodes.

Coarsening Strategies. Several algorithms have been proposed to identify which nodes of the grid are "coarse" (to be included in the next grid level), and which are "fine". This split is used to create the next grid level, but also forms vital information in the construction of the operators which interpolate values back between the coarse and fine grids.

Examples are the "classical" Ruge-Stüben coarsening [28], Cleary-Luby-Jones-Plassman (CLJP) [6], and parallel coarsenings which lead to lower operator complexity, such as Parallel Modified Independent Set (PMIS) and HMIS [7]. The aggressiveness of the coarsening influences the size of the next-coarsest grid (and thus the run-time of the final algorithm), but also impacts the subsequent choice of interpolation operators.

Interpolation and the Construction of the Prolongation Operator. Between two grid levels, the error propagation matrix is given by

$$M = S^{\sigma_2}(I - P[P^T AP]^{-1}P^T A)S^{\sigma_1} \tag{2}$$

where the S terms represent error propagation matrices of the pre- and post-smoothing relaxation sweeps.

The asymptotic convergence rate of AMG is determined by the spectral radius of this error propagation matrix $\rho(M)$. After the pre- and post-smoother are chosen, $\rho(M)$ depends only on P. While a good P is one that results in a small spectral radius for M, it should also be sparse for computational efficiency.

We note that the restriction operator which maps from fine to coarse grids is often chosen such that $R = P^T$.

The methods we consider in Sect. 3 will aim to improve on the weights in a candidate P.

Use of AMG as a Preconditioner. The multilevel nature of AMG means that each iteration is much more expensive to execute than an iteration of a single-level Krylov solver. As a result, instead of being used as a standalone solver, AMG is frequently used as a preconditioner to improve the convergence of Krylov subspace methods. In this task, only a few AMG iterations are run to get an approximate solution, and the setup cost of AMG (including building P) becomes more important than the cost of AMG iterations to solve the system.

2.2 A Dataset of Sparse Systems from 3D Unstructured Meshes

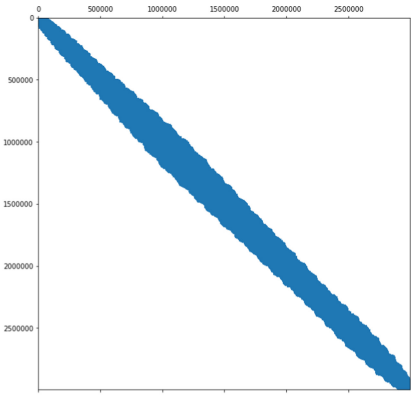
To evaluate the usefulness of newly proposed methods, we build a large dataset of sparse matrices captured from simple FEM fluid simulations on a diverse set of geometries. Our search for existing open datasets of geometries to leverage as a starting point led us to Zhou and Jacobson's 2016 Thingi10K dataset [38], which contains 10,000 3D printing models. Similar sources of 3D shape datasets exist in the ModelNet [34] and ShapeNet [4] datasets. However, Thingi10K was chosen since it has already been used as an evaluation set for the FTetWild mesher [19], which transforms "triangle soup" input meshes into high quality tetrahedral meshes that are valid for use with FEM simulations.

To generate our dataset, we use the FEniCS [23] framework with the 10,000 high quality versions of the Thingi10K meshes after FTetWild processing, and implement a simple 3D Navier-Stokes simulation using P2-P1 (Taylor-Hood) finite elements with a time-varying pressure condition, and Dirichlet boundary conditions. At each timestep, the solver must solve three large linear systems representing the velocity update, pressure correction, and velocity correction steps. After a few timesteps, we capture these linear systems to files, giving us 30,000 test matrices.

This process can be re-run with a variety of boundary and initial conditions, different PDEs with various coefficients, different FTetWild triangle densities, and different solvers (e.g. finite volume method simulations) to easily generate many other candidate sparse linear systems to evaluate the accuracy of acceleration methods.

An example of a mesh from the Thingi10K dataset is shown in Fig. 1. Here we show Item 47251 ("Living Brain Right Part"). The geometry is transformed using FTetWild then used as the basis for a simple FEM simulation with 438,897 degrees of freedom. We show a rendering of the original mesh (252,204 faces), and a visualization of the corresponding sparse matrix from the velocity update step of our simulation. The square matrix is challenging at 2,995,242 rows, but is also 99.997% sparse. While this is still orders of magnitude smaller than problem sizes in industrial applications, it is more representative of the problems we wish to solve than the small test problems used in [24] (64–400,000 rows).

(a) Rendering of the mesh geometry

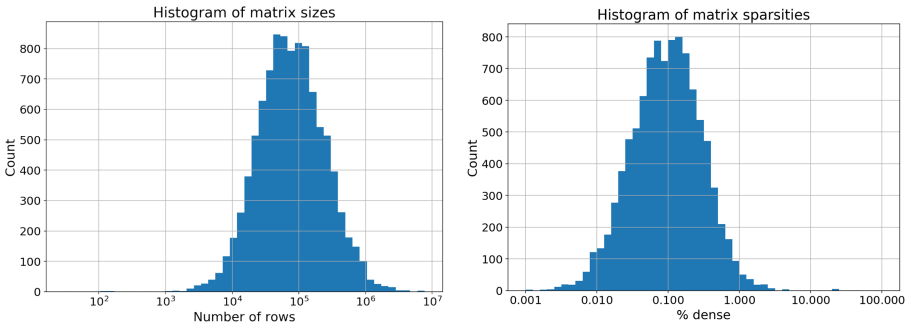


(b) Corresponding sparse matrix

Fig. 1. Item 47251 *Living Brain Right Part* from the Thingi10K dataset, with a visualisation of the corresponding sparse matrix from a FEM fluid simulation using this geometry.

Figure 2 gives an overview of the matrix sizes and sparsity of the matrices in the dataset (using the default FTetWild triangle density). As we will discuss in

Sect. 3.2, these sparse matrices are also much larger than those used in benchmark datasets for common graph neural network frameworks.



(a) Distribution of matrix sizes (b) Distribution of matrix sparsities

Fig. 2. Distribution of matrix sizes and sparsities in the dataset (velocity projection step only)

3 Overview of the Recently Proposed Methods

The earliest work we are aware of which tries to predict better prolongation operators for AMG using machine learning is the Deep Multigrid Method proposed by Katrutsa, Daulbaev and Oseledets [21], which they demonstrate only on small structured grid problems for PDEs in 1D. The method is not practical because it requires training a new deep neural network for every new matrix A. We did not apply it in our work.

For our work, we were specifically attracted to the approaches in Luz et al. [24] and the work preceding it in Greenfeld et al. [16], which try improving on the values in a candidate P produced by some existing method. Since these methods use previously computed coarsening and prolongation sparsities, they can leverage the decades of work in existing AMG solvers.

Learning to Minimize $\rho(M)$. In all three of these works, the problem of predicting a good P is recast as a learning problem where the aim is to minimize $\rho(M)$, the spectral radius of the two-grid error matrix from Eq. 2.

Any algorithm that tries to minimise $\rho(M)$ directly faces a difficult time: evaluating $\rho(M)$ means inverting the large $(P^T AP)^{-1}$ term in Eq. 2, and subsequently performing a very expensive $(\mathcal{O}(n^3))$ eigen-decomposition with n extremely large. To make matters worse, neither TensorFlow nor PyTorch (the two dominant DL frameworks today) currently support sparse matrix inversion, and converting to dense representations is infeasible for problems of this size.

Instead, Katrutsa, Daulbaev and Oseledets approximate the spectral radius using Gelfand's formula [21], and develop an efficient, unbiased stochastic estimator for this approximation. It still requires evaluating norms of M, and thus computing the expensive $(P^T AP)^{-1}$ term.

In contrast, both [24] and [16] choose a proxy loss function: they minimize the upper bound for the spectral radius given by the Frobenius norm $\|M\|_F^2$. To avoid the large cost of evaluating the $(P^T AP)^{-1}$ term directly, they restrict their training data distribution to specially synthesized block-circulant, block-diagonalized matrices, where the norm is the sum of the small diagonal blocks' norms. This allows efficient and stable calculation of the loss function, but severely restricts the distribution of input matrices available for training.

3.1 Deep Residual Feed-Forward Network for 2D Structured Grid Problems (Greenfeld et al. [16])

Our interest is in methods that work for 3D *unstructured* grids, but in this section we briefly discuss the work of Greenfeld et al. [16]. Their neural network model is for a 2D structured grid discretized using a 3×3 stencil. However, it lays important groundwork for the unstructured grid method which follows.

Input and Output. The proposed deep neural network aims to improve on values in a candidate prolongation matrix as determined by an existing interpolation algorithm, by only using information local to each coarse point. The same network is used to predict P at all levels of the grid.

In addition to inference against the model, there is non-trivial pre- and post-processing to assemble the inputs and interpret the neural network's outputs.

As input, for each coarse point in the grid, we pass to the network the equivalent point in the fine grid and its four nearest neighbors, using the 3×3 stencil of each of these 5 points, resulting in 45 real values which are concatenated and flattened to build an input in $\mathbb{R}^{45}$.

The output of the network is fixed to be in $\mathbb{R}^4$ and represents the weightings of the coarse point to the fine grid points to the north, south, east and west. Remaining directions' weightings are then solved algebraically. In the specific problem formulation the authors choose, the sparsity of the columns in P is fixed to allow at most 9 non-zeros, and the maximum distance to which coarse points can contribute to fine points is fixed at 1.

Despite its limitations, this network is more flexible than earlier proposals for PDE solvers such as the one in Tang [30]: it works for grids of any size without re-training, as long as the stencil size of the problem is the same. The authors also show some generalization capability when used with different boundary conditions and distributions of coefficients for the underlying PDE.

However, this approach involves very tight integration with the problem being solved, and means training entirely different networks for different discretizations (consider that when using a 3D 27-point stencil instead of a 2D 9-point stencil, the concatenated, flattened inputs will be in $\mathbb{R}^{189}$ instead of $\mathbb{R}^{45}$), and needs an entirely different formulation to use longer-range interpolation strategies. It is

also limited to structured grids. So we look to another proposal to tackle the unstructured problems of interest in our dataset.

3.2 Graph Neural Networks for Unstructured Problems (Luz et al. [24])

Seeking to extend the work of Greenfeld et al. to unstructured problems, Luz et al. develop an approach in [24] which uses the Graph Neural Network shown in Fig. 3.

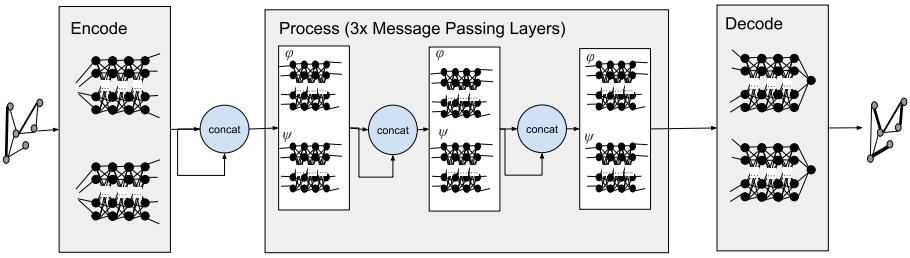


Fig. 3. The graph neural network used in [24].

GNNs are a recent kind of machine learning model that can deal with general unstructured input. A common unifying framework for reasoning about graph networks is presented by Bataglia et al. in [1], who also developed the DeepMind GraphNets library that Luz et al. used for the implementation of this model.

Luz et al. learn to predict P using a non-recurrent encode-process-decode architecture, in which

- the *encode* block learns a representation of the input edge and node features in a higher-dimensional latent space ($\mathbb{R}^{64}$). It has two 4-layer dense feed-forward neural networks to learn this function for edges and nodes independently.
- the *process* block consists of 3 message-passing layers. In graph networks, message passing layers (Gilmer et al. [14]) are a graph generalization of spatial convolution, and are essential for learning representations. Message passing layers produce a "message" for each edge by evaluating some learned function ϕ, taking as input the edge feature and the node features of the edge's incident nodes. Then, for each node, we aggregate the messages from its edges using another learned function ψ which takes as input the sum reduction of the incoming messages. In graph neural networks, the learnable functions ϕ and ψ are implemented using some choice of neural network. Luz, et al. use 4-layer dense feed-forward networks, but also choose to concatenate each message layer's input with the previous layer's input[1].

[1] Plus we note that a bug in the original authors' implementation results in a superfluous concatenation of the encoded input with itself before being fed to any message passing layers.

– the *decode* block translates the high-dimensional latent node and edge representations back into a single real value per edge using two independent 4-layer feed-forward networks.

This model has 286,146 learnable parameters.

In addition to the model, there are non-trivial pre-processing and post-processing tasks to translate between the sparse tensors in the problem domain and the data structure (a "GraphsTuple") used by the GraphNets framework.

Similar to the Greenfeld et al. model in Sect. 3.1, to prepare the input to the network we need: the list of coarse nodes from an existing coarsening method, a candidate prolongation matrix from an existing method, and the sparse matrix A. These are transformed into a GraphsTuple structure in which nodes are one-hot encoded to indicate whether they are coarse or fine, and edge values are concatenated with a one-hot encoding indicating whether they are included in the candidate P's sparsity pattern. Note that the size of the graph structure depends on both the size and the sparsity of A.

The output from the graph network is another GraphsTuple, from which the final P must be reconstructed. The original coarse nodes list is used to select the columns of P, and the baseline P is used to re-impose the sparsity pattern (i.e. only selecting those edge values from the graph network output).

The current implementation uses the DeepMind GraphNets library, which in turn is based on the Sonnet library that builds on the well-known TensorFlow ML framework, allowing the implementation to use TensorFlow's optimised kernels for sparse and dense Tensor operations on various devices, and benefiting from TensorFlow's extensive ecosystem of tooling. However, we note that other graph network libraries exist which are more actively maintained and report better performance, such as DGL [31].

4 Results

4.1 For the Model in Greenfeld et al.

Faster Training with Simpler Models. The authors of [16] choose a 100-wide, 100-deep residual feed-forward network with over 10^6 trainable parameters to learn the $f : \mathbb{R}^{45} \rightarrow \mathbb{R}^4$ mapping, without justifying why such an expressive network might be necessary (although they state in their conclusions that future work should consider simpler models). The network as proposed in [16] took almost a day to train on a high-end (Intel Xeon 8168 48-core) CPU, or several hours on a GPU – a prohibitive upfront cost for a method so tightly integrated with the specific problem being solved, considering that the problem sizes in [16] only take seconds to solve using a traditional solver.

Pursuing simpler, shallower neural network models is attractive because they would not only allow for a faster forward pass and much less computation during inference, but also consume less memory, and require fewer weights to be loaded from a serialized description of the model.

We retrained two much simpler network architectures: a simple 4-layer encoder-decoder network (layer widths 100, 50, 30, 50 with ReLu activations) with 12,998 learnable parameters and a shallow 2-layer multi-layer perceptron (layer widths 20, 10 with ReLu activations) with 1,038 learnable parameters. Layer widths were chosen to result in networks with approximately two and three orders of magnitude fewer learnable parameters than that proposed in [16], respectively.

Informed by results showing that incorporating large learning rates aids rapid learning convergence (Wilson et al. [32]), we also introduced an exponentially decaying learning rate schedule with a large initial value ($\epsilon = 0.1$, decay rate 0.95) to replace the authors' approach of only setting a small initial learning rate ($\epsilon = 1.2 \times 10^{-5}$). When used with the common "early-stopping" technique [27], these changes can reduce training from hours to minutes on a CPU, yet we still achieve very good prediction accuracy. The simple networks continue to outperform the classical Ruge-Stüben interpolation the authors of [16] use as a baseline comparison, although not as well as the original network, and the simplest network's results do not generalize to larger matrix sizes (see Table 1).

Table 1. Performance of simpler network architectures applied to the work in [16], indicating the percentage of cases where the model's predicted P resulted in convergence in fewer AMG V-cycles than a baseline Ruge-Stüben solver to solve the 2D FEM problem.

Grid size	Original from [16]	4 layer enc-dec	2-layer MLP
32×32	83%	93%	96%
64×64	92%	90%	88%
128×128	91%	88%	78%
256×256	84%	82%	78%
512×512	81%	86%	68%
1024×1024	83%	80%	52%

Integration and Parallelization. The design of the model in [16] allows it to be used in a parallel, distributed-memory setting. Gathering the stencil of surrounding nodes uses mostly local values and communication patterns very similar to those required during coarsening and classical interpolation, and allows extensive re-use of values already in a distributed node's memory. Similarly, since the network considers each input in isolation, it is possible to have several distributed instances of the model to query (e.g. one per compute node), and we can batch inputs to the network for efficient computation. Finally, the simplest network we found has a very light computational profile (two layers of matrix multiplication and activation function computation) and not many weights to load during model deserialization, making it feasible to use the model on the same compute

nodes where the coarsening and interpolation work happen. Models are agnostic to the size of the input grid, making them re-usable. Even though a different model may be needed for each problem (distribution, boundary conditions), having pre-trained model weights for a variety of problem settings available for download in a 'model zoo' might be feasible.

4.2 For the Model in Luz et al.

Scalability of Inference. First, we try to evaluate how well a model trained as described in [24] copes with the matrices in our dataset without modification.

A problem quickly becomes apparent: as shown in Fig. 4, when performing inference on a single node, the execution time of the forward pass of the graph network quickly becomes problematic for larger matrices, e.g. taking almost 5 min for a medium-sized $800,445 \times 800,445$ matrix in our test set on a 48-core Intel Skylake CPU. This inference represents only the first level of the grid's setup of P – in a real problem setting, subsequent levels still would need to make inferences in serial (for progressively smaller graphs) during the problem setup phase. For comparison the single-core pyAMG [2] solver using a classical direct framework for this same problem takes just 71.1s for the whole problem setup phase and 50.5s for the iterations, i.e. the traditional approach using a single core can solve the whole system in less time than just a small portion of the setup phase takes using the GNN model.

The time complexity of the GNN inference is linear in the size of the input network. Wu [35] derives the time complexity of similar message-passing Graph Convolutinal Network (GCN) layers as $\mathcal{O}(Kmd + Knd^2)$ with K the number of layers in the network, m the number of edges and n the number of nodes d is a fixed constant denoting the size of the hidden features. This is borne out by the linear scaling seen in Fig. 4, where deviation from the ideal slope is attributable to the slightly different sparsities of the matrices affecting the number of edges in the graph.

While the runtime is already problematic, a bigger problem is the memory usage of this network. In the default implementation, modest compute nodes with only 16GiB of RAM (or smaller GPUs) run out of memory at inputs of about $20,000$ nodes. The largest matrix we were able to use as input ($800,000$ nodes) on a more powerful compute node used approximately 400GiB of RAM to perform the query.

As with run-time, the memory required for inference in this network also grows linearly with the size of the network. Specifically, for the message passing layers we look to a result in Wu [35] which demonstrates the memory complexity for graph convolutional layers as $O(Knd + Kd^2)$, with K the number of layers in the network, n the number of nodes and d a fixed constant denoting the size of the hidden features. However, since the network architecture in [24] also *concatenates* outputs of each message passing layer as inputs to the next layer, this size of the latent space grows with each layer, with each edge feature consisting of 384

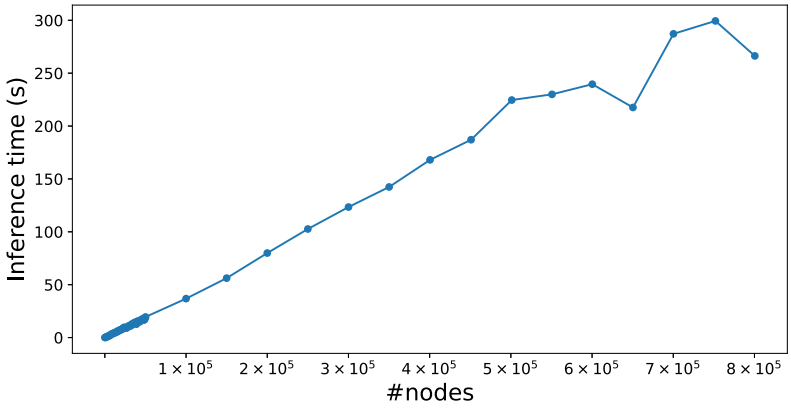


Fig. 4. Inference time against the graph network model in seconds on a 48 core Intel Xeon (Skylake) 8168 (2.7 GHz) CPU, for increasing graph sizes (excluding input and output processing). The largest problem size for which inference succeeded had 800,445 nodes and 49,609,863 edges. Larger problems ran out of memory (using in excess of 391 GiB of RAM)

`float64`'s after the last message passing layer! Luz et al. perform an ablation study which shows that omitting this concatenation slightly hurts the accuracy of the network.

Memory usage for exact inference in Graph Neural Networks today is known to be problematic [35]. Intermediate layers store the values for all nodes, making memory usage highly sensitive to the size of input graphs. The typical approach is to resort to stochastic sampling of a limited number of nodes or edges in a neighborhood [17], but our problem setting for predicting P makes this impossible (we need the edge values of all elements of P in our output). For exact inference against large networks on GPUs, it is possible to loop over the nodes and edges per layer in batches, and copy intermediate layer values back to CPU memory. But this approach requires a manual, device-specific implementation that reduces portability [8], and still assumes that the data fits in CPU memory.

Our situation is not unusual: we see other recent works reporting out-of-memory errors (OOMs) on modest network sizes (e.g. the performance comparison of various frameworks in [31]), and note that benchmarks for GNNs typically use small (<20,000 node) problem sizes, focus on training times and report single-node inference [10,33]. This unrepresentative state of affairs is being addressed by initiatives such as the recent Open Graph Benchmark dataset [18].

Of course, we can use further tricks to squeeze more performance from a single node, such as using mixed precision (`float32` accumulation and `float16` weights in the network) or reduced precision datatypes for the intermediate vectors instead of the `float64` widths used by Luz et al. (although we note that these are not supported by the versions of the frameworks used in their original implementation), and possibly reducing the dimensionality of the latent space (not tested in the authors' ablation study). Skillful application of techniques such

as inducing sparsity in the model through pruning, and the use of a DL compiler such as TVM [5] may reduce the computation and memory usage to allow for much larger single-node inferences. In addition, alternative frameworks like DGL have much better memory use owing to their use of fused message passing kernels that do not store the intermediate message values [31]. However, even with such mitigations, scaling to the large problem sizes with billions of nodes as required by industrial-scale problems is not feasible on a single inference node.

Distributed Inference. Industrial use cases involving billion-scale graphs, such as that reported by Ying et al. [37] on 18-billion edge graphs at Pinterest, have led to the development of large-scale frameworks. A survey of leading GNN frameworks (PyTorch Geometric [12], DGL [31], GraphNets [1], Spektral [15], Angel [20], Graph-Learn [36]) shows widespread support for distributed *training*, especially using data-parallel rather then model-parallel techniques, but distributed *inference* (where a single large input graph is distributed over many nodes) remains poorly supported. Our survey found support for distributed inference in recent versions of DGL, Angel, and Graph-Learn but the heavyweight process for initializing the servers and workers, determining graph partitions, and loading the distributed data efficiently (from shared storage) to build the distributed graph is overkill given that we only need to make a single inference during the AMG problem setup phase per multigrid level. We have not re-implemented the model in these frameworks as part of our work.

Model Input and Output Processing. In our integrations with scalable solvers such as HYPRE, we face the problem that the large sparse matrix A is partitioned over distributed compute nodes' memories. The transformation from this sparse matrix into a single "GraphsTuple" input for inference against the model using some standard serving framework can easily be implemented in a parallel, distributed fashion, but the interface between this distributed representation and the model framework is awkward: when using a serving framework, e.g. by initiating a request using the gRPC framework, we still need to send the whole graph, and by implication have gathered it into a single originating node's memory. Similarly, the response must be unmarshalled, the sparsity pattern of P re-imposed, and the resulting elements of P repartitioned to their distributed locations to continue with solver setup.

Testing Small-Graph Generalization Without Retraining. Despite training on a limited class of matrices that allow for cheap evaluation of the loss function, Luz et al. show some generalization capability of this network to other distributions and even different PDEs problems without retraining, including to a simple 2D FEM problem.

However, we note that the inputs to the network are also shaped by the coarsening strategies and interpolation methods used (they are used in the preprocessing and post-processing stages to impose a sparsity pattern). The authors of [24] choose classical Ruge-Stüben direct interpolation with the CLJP [6] coarsening and no Krylov acceleration as their baseline – a choice which sets the bar

for comparison quite low (it produces many more iterations and deeper hierarchies than a more aggressive coarsening and longer-range interpolation).

Classical AMG is designed for Hermitian positive definite matrices, and our FEM simulations are not guaranteed to produce these. Before testing how well a pre-trained model can generalize to our dataset, it is important to establish whether a baseline (non-ML) classical Ruge-Stüben solver, as configured with the coarsening and interpolation [24] used during training, can solve our systems. As a test, we use those "velocity projection" matrices with fewer than 25,000 rows, comprising 1,624 matrices from the dataset.

The solver is configured as in [24] to perform V-cycles using CLJP coarsening, a classical algebraic strength metric with $\theta = 0.25$, a single symmetric sweep of Gauss-Seidel pre- and post-smoothing, and a limit of 12 on the maximum number of levels. With this solver configuration, only 69.2% of the linear systems in the sampled dataset converge in under 500 iterations. Better results can be expected by using other options, such as smoothed aggregation coarsening with GMRES acceleration.

In contrast, when using the model from [24] to predict values of P, we find that only 17.7% of these systems converge in under 500 iterations. Even in cases where both the model and the baseline converge, the model's predictions result in fewer iterations to convergence only 1.2% of the time. It has not been able to generalize to the PDE and shapes in our dataset, even for the smallest matrices.

Infeasibility of Retraining or Fine-Tuning with Our Dataset. Lastly, we note that the prohibitive cost of evaluating the loss function on large matrices means we cannot use our dataset to retrain or fine-tune the network as it is currently proposed. Since TensorFlow does not currently support sparse inversion of matrices, the large $P^T AP$ matrices must be converted to dense matrices before matrix inversion, for which they are infeasibly large.

5 Conclusion

To date, the work of Luz, et al. [24] remains the only work we are aware of in this domain that supports unstructured grid problems. The proposed model has appealing 'black box' properties that mean the solver implementation is separated from the problem discretization, and the model can be used to complement the decades of work optimizing AMG solvers. However, we find that the model (as currently implemented) cannot scale usefully to even moderately sized matrices in our dataset. Although the situation may be improved by more optimization and re-implementation in a more performant framework, single-node inference cannot scale to the billions of nodes required for current applications. Yet unfortunately the current heavyweight setup for distributed graph inference frameworks is not suitable for this problem either: the nature of AMG means that inferences are infrequent (once per level per linear system) and necessarily sequential (cannot be batched for all the levels). But our work with the proposal from Greenfeld et al. shows that there may be hope for simpler models to achieve

good results on these problems, and these simpler networks are currently more amenable to deployment for inference in a distributed setting.

Computation for the coarsening and the classical interpolation must be carried out *in addition* and strictly prior to invoking the ML methods we considered, i.e. the run-time for the setup phase is guaranteed to be substantially longer. This only pays off if the cost of the additional inferences is recovered by run-time savings from substantially fewer iterations. For the small subset of our dataset with reasonable run-times, this was not the case.

Our hope had been to extend this approach to testing some of the very large models we are considering (with billions of degrees of freedom), but given the limitations of scaling the Graph Neural Network approach with existing implementations, we terminated our investigation early. Future work is required before these methods are ready for general application to real-world problems.

Future Work. We are investigating the use of DLPack [9], a proposal for an in-memory tensor structure that allows no-copy tensor sharing between DL networks. It already has support in PyTorch, TensorFlow, DGL and PetSc, potentially enabling easier integration of traditional HPC and DL tools.

Re-implementing the model of Luz et al. in a more actively developed framework such as DGL will allow more scope for optimization using techniques such as reduced precision and weight quantization, and can allow evaluating whether stochastic neighborhood sampling can be made to work with this problem and relieve memory pressure.

Adapting the model in [24] for smoothed aggregation and longer-range interpolation, and perhaps combining it with preconditioners for FEM such as AMGe [3], may prove to be more successful for the problems in our dataset.

Lastly, we are considering a hybrid of the approaches in [16] and [24], in which the distance-4 neighborhoods for each point are gathered at their distributed nodes and passed to local, simpler graph networks which operate only on the small subgraphs, allowing the GNN approach to scale better.

A Dataset

Code and instructions for reproducing our dataset and training the models can be found at https://github.com/UoB-HPC/scaling-ml-approaches-to-amg.

References

1. Battaglia, P.W., et al.: Relational inductive biases, deep learning, and graph networks. arXiv preprint arXiv:1806.01261 (2018)
2. Bell, W., Olson, L., Schroder, J.: PyAMG: algebraic multigrid solvers in Python (2011). https://github.com/pyamg/pyamg. Accessed 01 June 2021
3. Brezina, M., et al.: Algebraic multigrid based on element interpolation (AMGE). SIAM J. Sci. Comput. **22**(5), 1570–1592 (2001)
4. Chang, A.X., et al.: ShapeNet: an information-rich 3D model repository. arXiv preprint arXiv:1512.03012 (2015)

5. Chen, T., et al.: TVM: end-to-end optimization stack for deep learning, pp. 11–20. arXiv preprint arXiv:1802.04799 (2018)
6. Cleary, A.J., Falgout, R.D., Henson, V.E., Jones, J.E.: Coarse-grid selection for parallel algebraic multigrid. In: Ferreira, A., Rolim, J., Simon, H., Teng, S.-H. (eds.) IRREGULAR 1998. LNCS, vol. 1457, pp. 104–115. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0018531
7. De Sterck, H., Yang, U.M., Heys, J.J.: Reducing complexity in parallel algebraic multigrid preconditioners. SIAM J. Matrix Anal. Appl. **27**(4), 1019–1039 (2006)
8. DGL v0.6.1 user guide: exact inference against large graphs. https://docs.dgl.ai/en/0.6.x/guide/minibatch-inference.html (2018). Accessed 01 June 2021
9. DLPack: Open in memory tensor structure. https://github.com/dmlc/dlpack (2017). Accessed 06 June 2021
10. Dwivedi, V.P., Joshi, C.K., Laurent, T., Bengio, Y., Bresson, X.: Benchmarking graph neural networks (2020)
11. Falgout, R.D., Yang, U.M.: *hypre*: a library of high performance preconditioners. In: Sloot, P.M.A., Hoekstra, A.G., Tan, C.J.K., Dongarra, J.J. (eds.) ICCS 2002. LNCS, vol. 2331, pp. 632–641. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-47789-6_66
12. Fey, M., Lenssen, J.E.: Fast graph representation learning with PyTorch geometric. arXiv preprint arXiv:1903.02428 (2019)
13. Gee, M., Siefert, C., Hu, J., Tuminaro, R., Sala, M.: ML 5.0 smoothed aggregation user's guide. Technical Report SAND2006-2649, Sandia National Laboratories (2006)
14. Gilmer, J., Schoenholz, S.S., Riley, P.F., Vinyals, O., Dahl, G.E.: Neural message passing for quantum chemistry. In: ICML 2017, pp. 1263–1272. PMLR (2017)
15. Grattarola, D., Alippi, C.: Graph neural networks in TensorFlow and Keras with Spektral [application notes]. IEEE Comput. Intell. Mag. **16**(1), 99–106 (2021)
16. Greenfeld, D., Galun, M., Kimmel, R., Yavneh, I., Basri, R.: Learning to optimize multigrid PDE Solvers. In: 36th ICML 2019 June, pp. 4305–4316, February 2019
17. Hamilton, W.L., Ying, R., Leskovec, J.: Inductive representation learning on large graphs. arXiv preprint arXiv:1706.02216 (2017)
18. Hu, W., et al.: Open graph benchmark: Datasets for machine learning on graphs. arXiv preprint arXiv:2005.00687 (2020)
19. Hu, Y., Schneider, T., Wang, B., Zorin, D., Panozzo, D.: Fast tetrahedral meshing in the wild. ACM Trans. Graph. **39**(4) (2020). https://doi.org/10.1145/3386569.3392385
20. Jiang, J., Yu, L., Jiang, J., Liu, Y., Cui, B.: Angel: a new large-scale machine learning system. Natl. Sci. Rev. **5**(2), 216–236 (2018)
21. Katrutsa, A., Daulbaev, T., Oseledets, I.: Deep Multigrid: learning prolongation and restriction matrices. arXiv preprint arXiv:1711.03825 (2017)
22. Katsiapis, K., et al.: Towards ML engineering: a brief history of tensorflow extended (TFX). arXiv preprint arXiv:2010.02013 (2020)
23. Logg, A., Mardal, K.A., Wells, G.: Automated solution of differential equations by the finite element method: The FEniCS Book, vol. 84. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-23099-8
24. Luz, I., Galun, M., Maron, H., Basri, R., Yavneh, I.: Learning algebraic multigrid using graph neural networks. In: PMLR, pp. 6489–6499, November 2020
25. Open neural network exchange: The open standard for machine learning interoperability. https://www.onnx.ai. Accessed 04 June 2021
26. Paleyes, A., Urma, R.G., Lawrence, N.D.: Challenges in deploying machine learning: a survey of case studies. arXiv preprint arXiv:2011.09926 (2020)

27. Prechelt, L.: Early stopping—But when? In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) Neural Networks: Tricks of the Trade. LNCS, vol. 7700, pp. 53–67. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35289-8_5
28. Ruge, J.W., Stüben, K.: Algebraic multigrid. In: Multigrid Methods, pp. 73–130. SIAM (1987)
29. Stüben, K.: A review of algebraic multigrid. J. Comput. Appl. Math. **128**(1), 281–309 (2001). https://doi.org/10.1016/S0377-0427(00)00516-1. Numerical Analysis 2000. Vol. VII: Partial Differential Equations
30. Tang, W., et al.: Study on a Poisson's equation solver based on deep learning technique. In: 2017 IEEE EDAPS, pp. 1–3. IEEE (2017)
31. Wang, M., et al.: Deep graph library: A graph-centric, highly-performant package for graph neural networks. arXiv preprint arXiv:1909.01315 (2019)
32. Wilson, A.C., Roelofs, R., Stern, M., Srebro, N., Recht, B.: The marginal value of adaptive gradient methods in machine learning. arXiv preprint arXiv:1705.08292 (2017)
33. Wu, J., Sun, J., Sun, H., Sun, G.: Performance analysis of graph neural network frameworks. In: Proceedings - ISPASS 2021, pp. 118–127 (2021). https://doi.org/10.1109/ISPASS51385.2021.00029
34. Wu, Z., et al.: 3D shapeNets: a deep representation for volumetric shapes. In: Proceedings of the IEEE CVPR, pp. 1912–1920 (2015)
35. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Philip, S.Y.: A comprehensive survey on graph neural networks. IEEE Trans. Neural Netw. Learn. Syst. **32**, 4–24 (2020)
36. Yang, H.: AliGraph: a comprehensive graph neural network platform. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 3165–3166 (2019)
37. Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W.L., Leskovec, J.: Graph convolutional neural networks for web-scale recommender systems. Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, July 2018. https://doi.org/10.1145/3219819.3219890
38. Zhou, Q., Jacobson, A.: Thingi10K: a dataset of 10,000 3D-printing models. arXiv preprint arXiv:1605.04797 (2016)