

# ***TorchBraid: High-Performance Layer-Parallel Training of DNNs with MPI and GPU Acceleration***

---

**Prof. Jacob B. Schroder**

*University of New Mexico*

*Dept. of Mathematics and Statistics*



## **Collaborators:**

E. C. Cyr, J. Hahne, N. Gauger, S. Günther, M. Jiang, A. Kopanicakova, R. Krause, N. S. Moore, L. Ruthotto, M. Salvado, B. S. Southworth, D. A. Vargas

**Precond 24: International Conference On Preconditioning Techniques For Scientific and Industrial Applications**

June 12<sup>th</sup>, 2024

# Outline

- Need for parallelism driven by DNNs
  - Consider various forms of parallelism
  - Leverage wealth of parallel algorithms available from the CS&E community
- Deep neural networks
  - Interpret as an ODE (ResNets, Transformers, GRU)
- Layer-parallelism
  - Parallel-in-time applied to layers-as-time-steps
- Results
  - Code (TorchBraid)
  - Combine with coarsening inside layer (spatial coarsening)
  - Combine with data-parallelism
  - Tiny ImageNet: Combine with pooling and batch-norm
  - Transformers

# Need for parallelism driven by DNNs

- Deep neural networks (DNNs) routinely used to solve learning tasks with state-of-the-art accuracies
  - Image classification and recognition, natural language processing, self-driving cars, healthcare, ...
- However, the training time and cost is increasing
- ChatGPT-4 likely took month(s) to train using *many* GPUs
- Other sophisticated deep learning architectures also have long training times
- But the results of deep learning can be impressive!

# But the results are impressive!



You

how do I get more parallelism for my deep neural network?



ChatGPT

Increasing parallelism in deep neural networks can lead to improved performance and faster training. Here are several strategies you can consider:

### 1. Model Architecture:

- Utilize parallelizable layers such as convolutional layers, which are inherently parallelizable due to their local receptive fields.
- Use architectures designed for parallel processing, such as residual networks (ResNets) or dense networks (DenseNets), which facilitate parallel computation across layers.

### 2. Hardware Utilization:

- Invest in hardware specifically designed for parallel computation, such as GPUs (Graphics Processing Units) or TPUs (Tensor Processing Units), which excel at parallel matrix operations commonly found in deep learning.
- Utilize multiple GPUs or TPUs to distribute computation across devices. Frameworks like TensorFlow and PyTorch support this through data parallelism and model parallelism techniques.

### 3. Data Parallelism:

- Divide your dataset into smaller batches and process them simultaneously across multiple devices. This is the most common form of parallelism and is supported by most deep learning frameworks.
- Ensure that your batch size is large enough to fully utilize the available hardware without causing memory issues.

### 4. Model Parallelism:

# But ChatGPT, you forgot about my favorite type of parallelism!

ChatGPT doesn't quite  
have all the details yet.



You

What about layer parallelism?



ChatGPT

Layer parallelism is a technique where different layers of a neural network are executed in parallel across multiple processing units. It's a form of model parallelism, where the model is divided into parts, and each part is processed independently. Here's how layer parallelism can be implemented:

## 1. Dividing Layers:

- Divide the layers of your neural network into groups, with each group containing a subset of layers.
- The division can be based on various criteria, such as layer type (e.g., convolutional layers, fully connected layers) or computational complexity.

## 2. Processing Layers in Parallel:

- Assign each group of layers to a separate processing unit (e.g., GPU or TPU).
- Ensure that the layers within each group do not have dependencies on each other, allowing them to be executed concurrently.

## 3. Communication between Layers:

- Implement mechanisms for communication between layers processed in parallel, as some layers may require input from preceding layers or provide output to subsequent layers.
- Minimize communication overhead by optimizing data transfer between processing units.

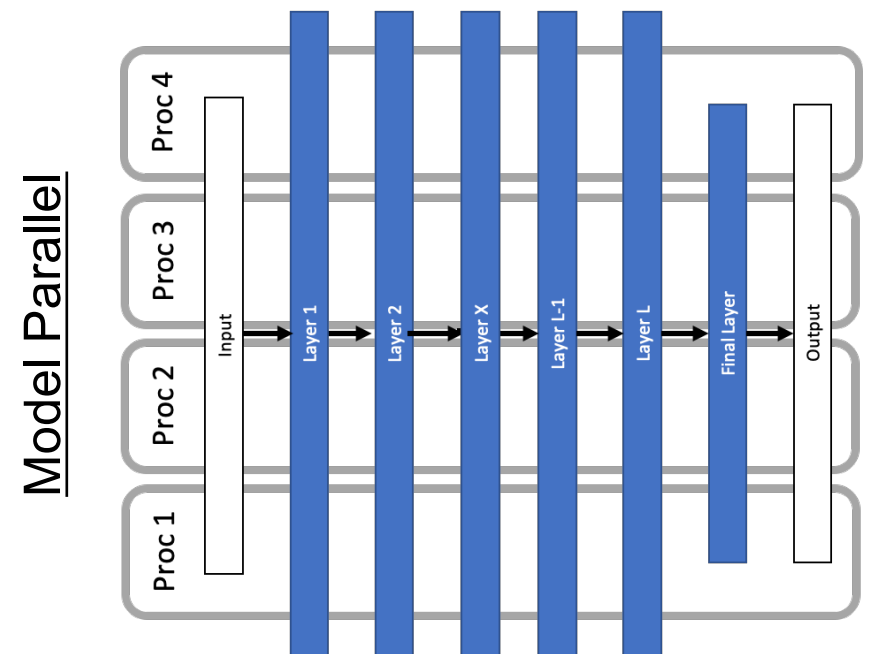
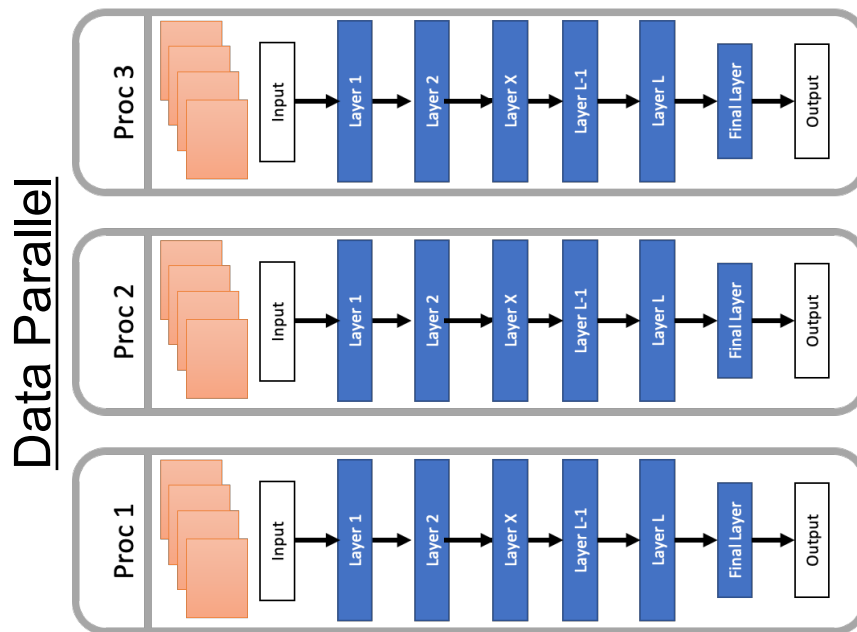
## 4. Synchronization:

- Ensure proper synchronization between processing units to maintain consistency during training.



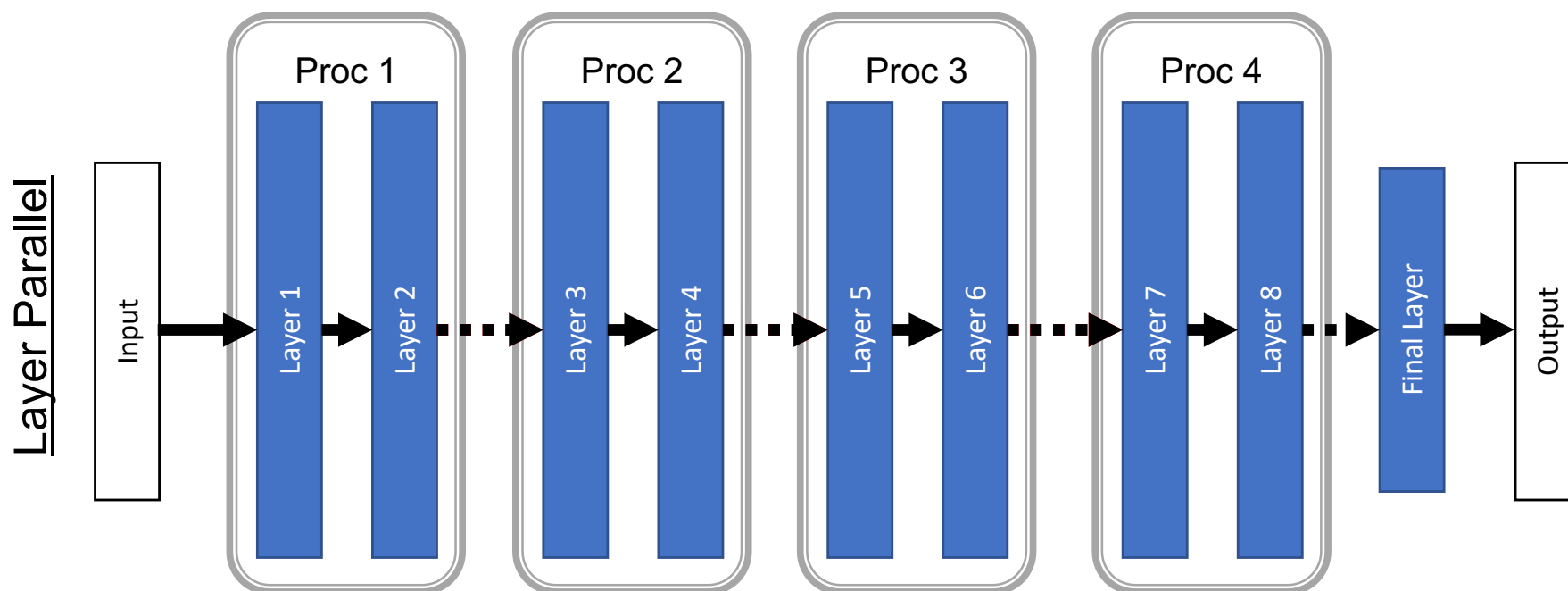
# Need for parallelism driven by DNNs

- With these long training times, we naturally search for parallelism
- Popular classic approaches for DNNs (as ChatGPT reminded us) include:
  - Data parallelism
  - Model (spatial) parallelism



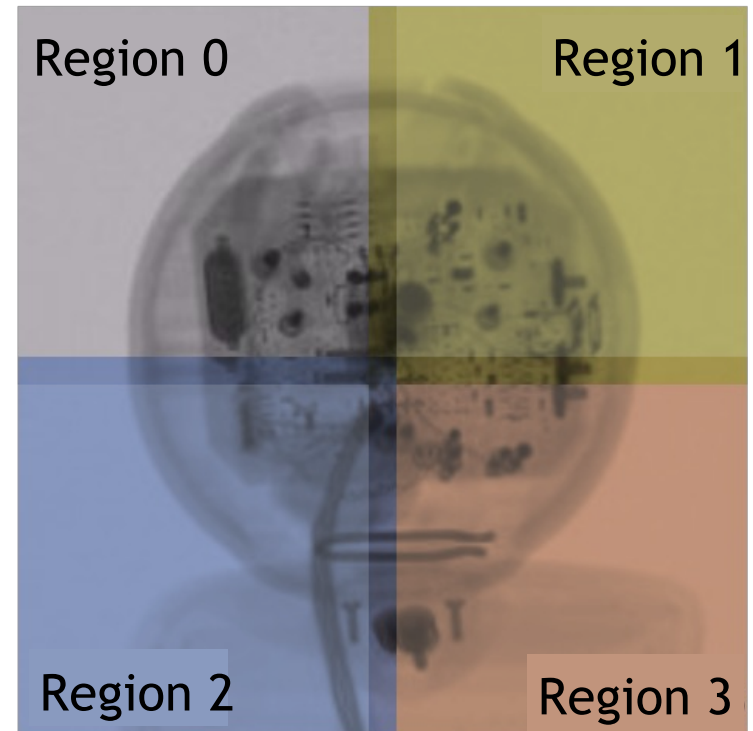
# Proposed Solution: Layer-Parallelism

- The ancient bargain: trade-off between parallelism and exactness
- Relax exactness for layer evolution in return for greater parallelism
  - Dotted arrows below become inexact (same is true for back-prop)
  - Trade accuracy for performance
  - But we don't need exact gradients (experiments tell us)
- Can be combined with other forms of parallelism, e.g., data parallelism



# Philosophically, we leverage parallel frameworks from CS&E to help accelerate DNN training

- This is an active field of research.
- Domain decomposition (patch-based) approaches
  - Apply (mostly) independent models to distinct regions of data (e.g., image), for parallelism
  - Combine region results in a variety of ways
  - Parallelism likely limited by largest reasonable patch size



Parallel decomposition for a CT scan (2D) of a webcam.

1. *Machine learning and domain decomposition methods -- a survey*, A. Klawonn, M.n Lanser, J. Weber, <https://arxiv.org/abs/2312.14050>
2. *Multifidelity DD-based PINNs for time-dependent problems*, A. Heinlein et al., <https://arxiv.org/abs/2401.07888>, 2024.
3. *Patch-based Convolutional Neural Network for Whole Slide Tissue Image Classification*, Hou et al. IEEE conference on computer vision and pattern recognition. 2016.
4. *Image scoring: Patch based CNN model for small or medium dataset*, C. Zhang, J. Cheng, 3rd IEEE International Conference on Computer and Communications, 2017.
5. *The Unreasonable Effectiveness of Patches in Deep Convolutional Kernels Methods*, L. Thiry et al., ICLR, 2021.
6. *A domain decomposition-based CNN-DNN architecture for model parallel training applied to image recognition problems*, A Klawonn, M Lanser, J Weber, <https://arxiv.org/abs/2312.14050>



# Philosophically, we leverage parallel frameworks from CS&E to help accelerate DNN training

- This is an active field of research.
- Multilevel optimization: target algorithmic speedup
  - Methods first used for PDE-constrained optimization
  - Construct coarse optimization problem(s) that can accelerate convergence for the finest-level optimization problem
  - Do forward and back-propagation on coarse- and fine-levels, interpolating parameters between levels
- Multilevel trust-region optimization methods<sup>1,3</sup>, for typical ML tasks like image classification
- MGOPT (FAS) approaches for optimization<sup>2</sup>, initially PDE approximation

1. *Globally convergent multilevel training of deep residual networks*, A. Kopaničáková, R. Krause, SISC, 2022.

2. *Multilevel-in-width training for deep neural network regression*, C. Ponce, R. Li, C. Mao, P. Vassilevski, NLAA, 2023.

3. *Multilevel minimization for deep residual networks*, L. Gaedke-Merzhäuser, A. Kopaničáková, R. Krause, ESAIM, 2021.

# Philosophically, we leverage parallel frameworks from CS&E to help accelerate DNN training

- This is an active field of research.
  - Related layer-parallel approach based on parallel solution to the linearized Jacobian of the optimization problem<sup>1</sup>
    1. Put your forward or backward network evaluation inside Newton's method
    2. Linearize the block lower bi-diagonal system
    3. Apply parallel cyclic reduction
    4. Rinse and repeat.
  - Very nice idea and generalizes to non-ODE networks
    - Current formulation suffers from "The main bottleneck for our DeepPCR implementation is represented by the decay in performance associated with the growth in size of the Jacobian blocks"
- ➔ These are just a few examples of current work, where CS&E inspires ML algorithms!

1. *DeepPCR: Parallelizing Sequential Operations in Neural Networks*, F. Danieli, M. Sarabia, X. Suau Cuadros, P. Rodriguez, L. Zappella, *Advances in Neural Information Processing Systems* 36, 2024.

# Outline

- Need for parallelism driven by DNNs
  - Consider various forms of parallelism
  - Leverage wealth of parallel algorithms available from the CS&E community
- Deep neural networks
  - Interpret as an ODE (ResNets, Transformers, GRU)
- Layer-parallelism
  - Parallel-in-time applied to layers-as-time-steps
- Results
  - Code (TorchBraid)
  - Combine with coarsening inside layer (spatial coarsening)
  - Combine with data-parallelism
  - Tiny ImageNet: Combine with pooling and batch-norm
  - Transformers

# Deep Neural Networks (DNNs)

- DNNs are routinely used for many tasks
  - However, training cost/times can be prohibitive (days, weeks, months...)
    - Due to the *many* forwards and backwards passes through the network
- **Goal:** parallelize, speed up training

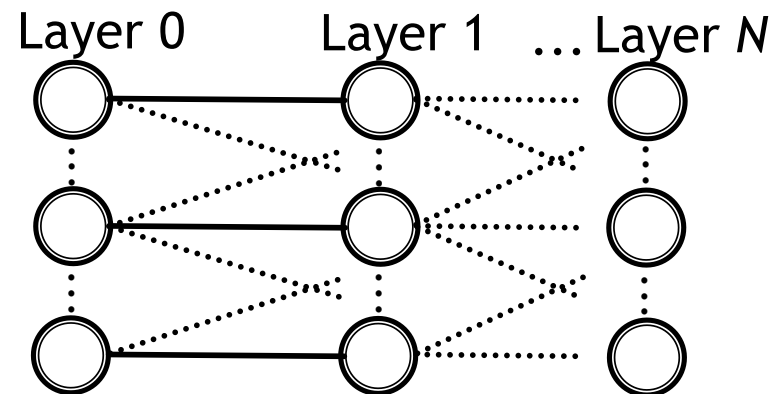
- Feed-forward network
  - Training pair:  $(y_{data}, c_{data})$
  - $W_n, b_n, y_n$  : Layer  $n$  weights, biases, state
  - **ResNet Propagation** (forward problem):

$$y_0 = y_{data}$$

$$y_{n+1} = y_n + F(W_n y_n + b_n) \quad \forall n = 0, \dots, N - 1$$

- Learning problem:

$$\min_{W_n, b_n} \text{Loss}(y_N, c_{data}) \quad \text{subject to above forward problem}$$



# ODE Perspective on Deep ResNets

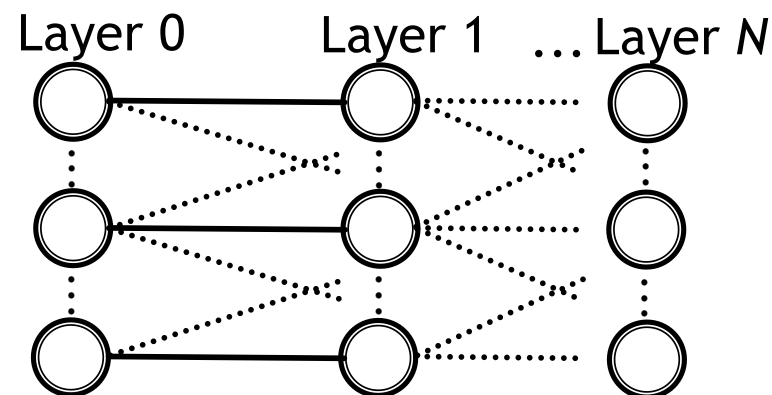
- DNNs are routinely used for many tasks
  - However, training cost/times can be prohibitive, in days or weeks
    - Due to the *many* forwards and backwards passes through the network
- **Goal:** parallelize, speed up training

- Feed-forward network
  - Training pair:  $(y_{data}, c_{data})$
  - $W_n, b_n, y_n$  : Layer  $n$  weights, biases, state
  - **ResNet Propagation** (forward problem):

$$y_0 = y_{data}$$

$$y_{n+1} = y_n + F(W_n y_n + b_n) \quad \forall n = 0, \dots, N - 1$$

Insert time-step parameter !



# ODE Perspective on Deep ResNets

- Some popular deep networks (e.g., ResNets) have an equivalence to time-stepping<sup>1,2</sup>
- Expand the ResNet architecture with a time-step parameter  $h$

$$y_0 = y_{data}$$

$$y_{n+1} = y_n + hF(W_n y_n + b_n) \quad \forall n = 0, \dots, N - 1$$

$$\Leftrightarrow$$

$$y(0) = y_{data}$$

$$\frac{dy(t)}{dt} = F(W(t)y(t) + b(t)), \quad \forall t \in (0, T)$$

- Training problem becomes

$$\min_{W(t), b(t)} \text{Loss}(y(T), c_{data}) \quad \text{subject to above ODE}$$

1. Haber, Ruthotto. *Stable Architectures for Deep Neural Networks*. Inverse Probl., 2017.  
2. Weinan, *A Proposal on Machine Learning via Dynamical Systems*, Comm. Math. Stat., 2017.

# ODE Perspective on Deep ResNets

- Some popular deep networks (e.g., ResNets) have an equivalence to time-stepping<sup>1,2</sup>
- Expand the ResNet architecture with a time-step parameter  $h$

$$y_0 = y_{data}$$

$$y_{n+1} = y_n + hF(W_n y_n + b_n) \quad \forall n = 0, \dots, N - 1$$

$\Leftrightarrow$

$$y(0) = y_{data}$$

$$\frac{dy(t)}{dt} = F(W(t)y(t) + b(t)), \quad \forall t \in (0, T)$$

Resnet  $\equiv$  Forward Euler discretization  
Backprop  $\equiv$  Discrete adjoint

- Training problem becomes

$$\min_{W(t), b(t)} \text{Loss}(y(T), c_{data}) \quad \text{subject to above ODE}$$

1. Haber, Ruthotto. *Stable Architectures for Deep Neural Networks*. Inverse Probl., 2017.  
2. Weinan, *A Proposal on Machine Learning via Dynamical Systems*, Comm. Math. Stat., 2017.

# ODE Perspective on Transformers

- Transformers are powerful (e.g., ChatGPT), where attention mechanisms capture long-range dependencies within sequences
- Transformers can also be extended to the layer-parallel setting
  - Ignoring pre- and post-processing
  - Transformers consist of encoder and decoder layers of the following form:

$$\mathbf{X}^{[n+1]} = \mathbf{X}^{[n]} + \mathbf{F}_{\text{enc}}(t_n, \mathbf{X}^{[n]}) \quad \mathbf{X}^{[n+1]} = \mathbf{X}^{[n]} + \mathbf{F}_{\text{dec}}(t_n, \mathbf{X}^{[n]})$$

$$\mathbf{F}_{\text{enc}}(t_n, x) := \varphi_1(x) + \varphi_2(x + \varphi_1(x)) \quad \mathbf{F}_{\text{dec}}(t_n, x) := \varphi_1(x) + \varphi_3(x + \varphi_1(x)) + \varphi_2(x + \varphi_1(x) + \varphi_3(x + \varphi_1(x)))$$

- Where  $\mathbf{X}^{[n]}$  is the network state at layer  $n$ , and  
 $\varphi_1 := \text{SA} \circ \text{LN}, \varphi_2 := \text{MLP} \circ \text{LN}, \varphi_3 := \text{CA} \circ \text{LN}$

for self-attention (SA), cross-attention (CA), and layer-norm (LN)

Related ODE Transformer works:

1. *Understanding and improving transformer from a multiparticle dynamic system point of view*, Y. Lu et al., 2019.  
<https://www.arxiv.org/abs/1906.02762>
2. *Stateful ODE-Nets using basis function expansions*, A. Queiruga et al., Advances in Neural Information Processing Systems, 2021.



# ODE Perspective on Transformers

- Transformers are powerful (e.g., ChatGPT), where attention mechanisms capture long-range dependencies within sequences
- Transformers can also be extended to the layer-parallel setting
  - Ignoring pre- and post-processing
  - Transformers consist of encoder and decoder layers of the following form:

$$\mathbf{X}^{[n+1]} = \mathbf{X}^{[n]} + \mathbf{F}_{\text{enc}}(t_n, \mathbf{X}^{[n]})$$

$$\mathbf{X}^{[n+1]} = \mathbf{X}^{[n]} + \mathbf{F}_{\text{dec}}(t_n, \mathbf{X}^{[n]})$$

$$\mathbf{F}_{\text{enc}}(t_n, x) := \varphi_1(x) + \varphi_2(x + \varphi_1(x))$$

$$\mathbf{F}_{\text{dec}}(t_n, x) := \varphi_1(x) + \varphi_3(x + \varphi_1(x)) +$$

Insert time-step parameter<sup>2</sup> !

Insert time-step parameter<sup>2</sup> !  $(x + \varphi_1(x))$

- Where  $\mathbf{X}^{[n]}$  is the network state at layer  $n$ , and

$$\varphi_1 := \text{SA} \circ \text{LN}, \varphi_2 := \text{MLP} \circ \text{LN}, \varphi_3 := \text{CA} \circ \text{LN}$$

for self-attention (SA), cross-attention (CA), and layer-norm (LN)

Related ODE Transformer works:

1. *Understanding and improving transformer from a multiparticle dynamic system point of view*, Y. Lu et al., 2019.

<https://www.arxiv.org/abs/1906.02762>

2. *Stateful ODE-Nets using basis function expansions*, A. Queiruga et al., Advances in Neural Information Processing Systems, 2021.

# ODE Perspective on Transformers

- Transformers are powerful (e.g., ChatGPT), where attention mechanisms capture long-range dependencies within sequences

- Transformers can also be extended to the layer-parallel setting

- Ignoring pre- and post-processing
- Transformers consist of encoder and decoder layers of the following form:

$$\mathbf{X}^{[n+1]} = \mathbf{X}^{[n]} + \mathbf{F}_{\text{enc}}(t_n, \mathbf{X}^{[n]})$$

$$\mathbf{X}^{[n+1]} = \mathbf{X}^{[n]} + \mathbf{F}_{\text{dec}}(t_n, \mathbf{X}^{[n]})$$

$$\mathbf{F}_{\text{enc}}(t_n, x) := \varphi_1(x) + \varphi_2(x + \varphi_1(x))$$

$$\mathbf{F}_{\text{dec}}(t_n, x) := \varphi_1(x) + \varphi_3(x + \varphi_1(x)) +$$

Insert time-step parameter<sup>2</sup> !

Insert time-step parameter<sup>2</sup> !  $(x + \varphi_1(x))$

- Where  $\mathbf{X}^{[n]}$  is the network state at layer  $n$ , and

$$\varphi_1 := \text{SA} \circ \text{LN}, \varphi_2 := \text{MLP} \circ \text{LN}, \varphi_3 := \text{CA} \circ \text{LN}$$

for self-attention (SA), cross-attention (CA), and layer-norm (LN)

- Learning problem remains unchanged

# ODE Perspective on GRU

- Gated recurrent unit (GRU) networks are powerful recurrent networks for sequence processing (NLP, video classification, ...)
- The basic GRU architecture at time  $t$  takes input data  $x_t$  and the previous hidden state  $h_{t-1}$  and computes a new hidden state  $h_t$

$$r_t = \sigma(W_{xr}x_t + b_{xr} + W_{hr}h_{t-1} + b_{hr})$$

$$z_t = \sigma(W_{xz}x_t + b_{xz} + W_{hz}h_{t-1} + b_{hz})$$

$$n_t = \phi(W_{xn}x_t + b_{xn} + r_t \odot (W_{hn}h_{t-1} + b_{hn}))$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot n_t.$$

- The implicit GRU formulation<sup>1</sup> changes the computation of  $h_t$  and achieves similar accuracy for considered test problems<sup>1</sup>

$$h_t = (1 + \Delta t(1 - z_t))^{-1} \odot (h_{t-1} + \Delta t(1 - z_t) \odot n_t).$$

- Learning problem remains unchanged

Original Layer-parallel GRU work:

1. *Parallel Training of GRU Networks with a Multi-Grid Solver for Long Sequences*, E. Moon and E. C. Cyr, ICLR 2022.

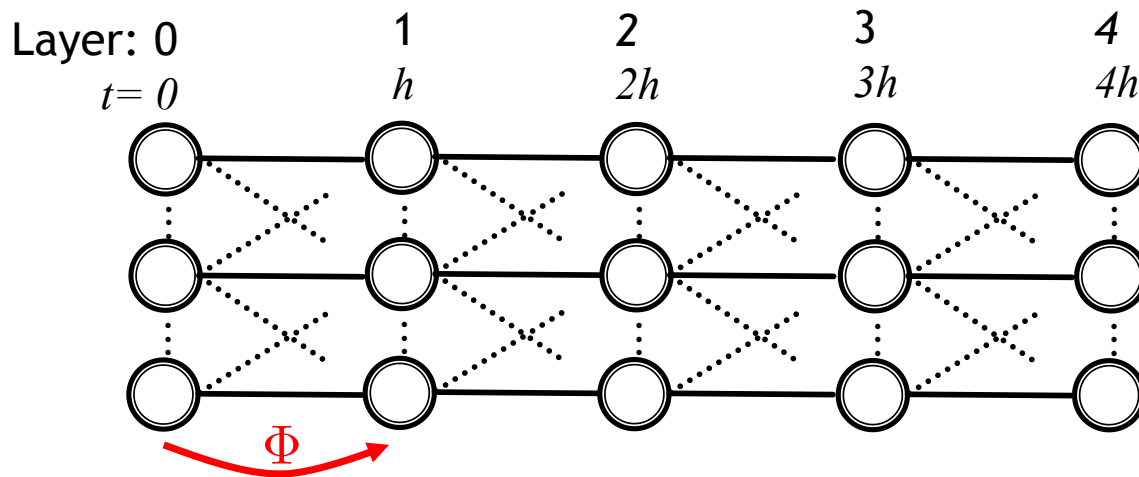
# Outline

- Need for parallelism driven by DNNs
  - Consider various forms of parallelism
  - Leverage wealth of parallel algorithms available from the CS&E community
- Deep neural networks
  - Interpret as an ODE (ResNets, Transformers, GRU)
- Layer-parallelism
  - Parallel-in-time applied to layers-as-time-steps
- Results
  - Code (TorchBraid)
  - Combine with coarsening inside layer (spatial coarsening)
  - Combine with data-parallelism
  - Tiny ImageNet: Combine with pooling and batch-norm
  - Transformers

# Parallel-in-time for DNNs

Layer  $i \equiv i$ -th time-step

- Network state  $y_i$  is "like" a spatial vector
- Distribute layers (time-steps) across  $N$  processors with parallel-in-time
- Forwards (backwards) network evaluations done iteratively in parallel
- Time-step  $\Phi$ : Propagation from layer  $n$  to layer  $n+1$

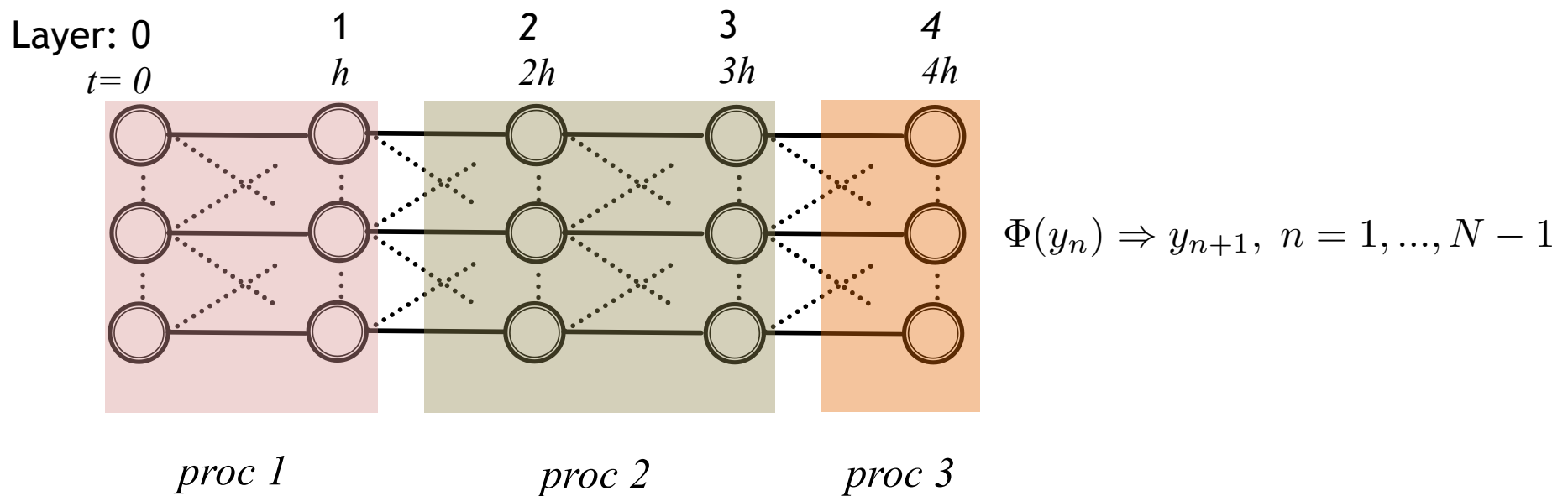


$$\Phi(y_n) \Rightarrow y_{n+1}, n = 1, \dots, N - 1$$

# Parallel-in-time for DNNs

Layer  $i \equiv i$ -th time-step

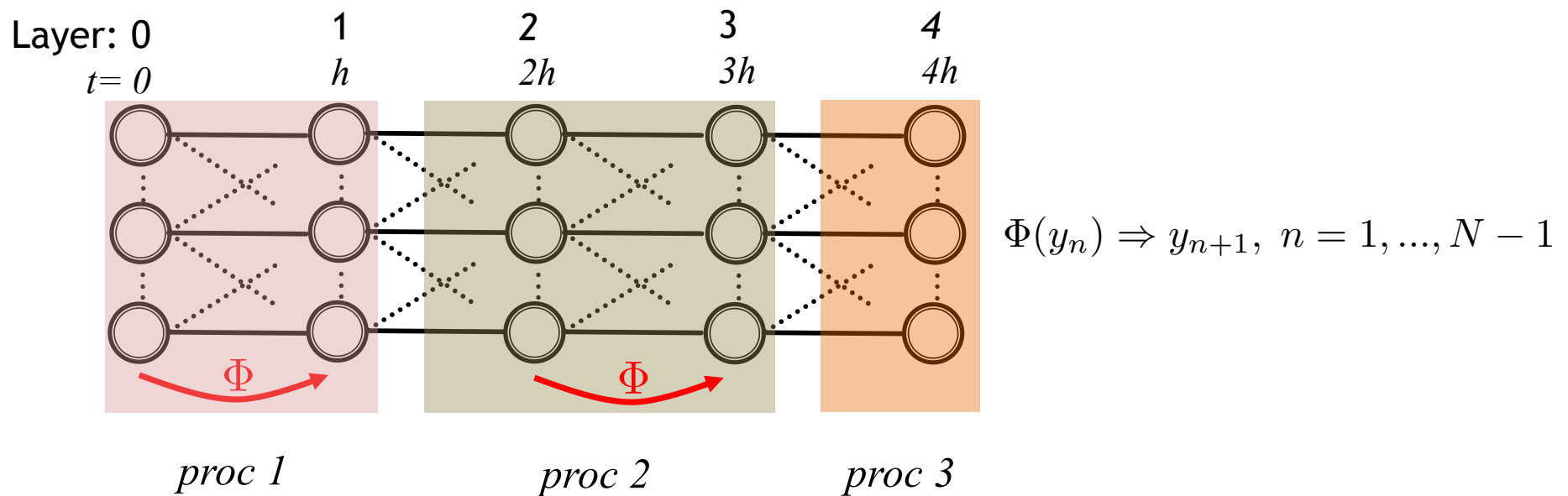
- Network state  $y_i$  is "like" a spatial vector
- Distribute layers (time-steps) across  $N$  processors with parallel-in-time
- Forwards (backwards) network evaluations done iteratively in parallel
- Time-step  $\Phi$  : Propagation from layer  $k$  to layer  $k+1$



# Parallel-in-time for DNNs

Layer  $i \equiv i$ -th time-step

- Network state  $y_i$  is "like" a spatial vector
- Distribute layers (time-steps) across  $N$  processors with parallel-in-time
- Forwards (backwards) network evaluations done iteratively in parallel
- Time-step  $\Phi$ : Propagation from layer  $k$  to layer  $k+1$



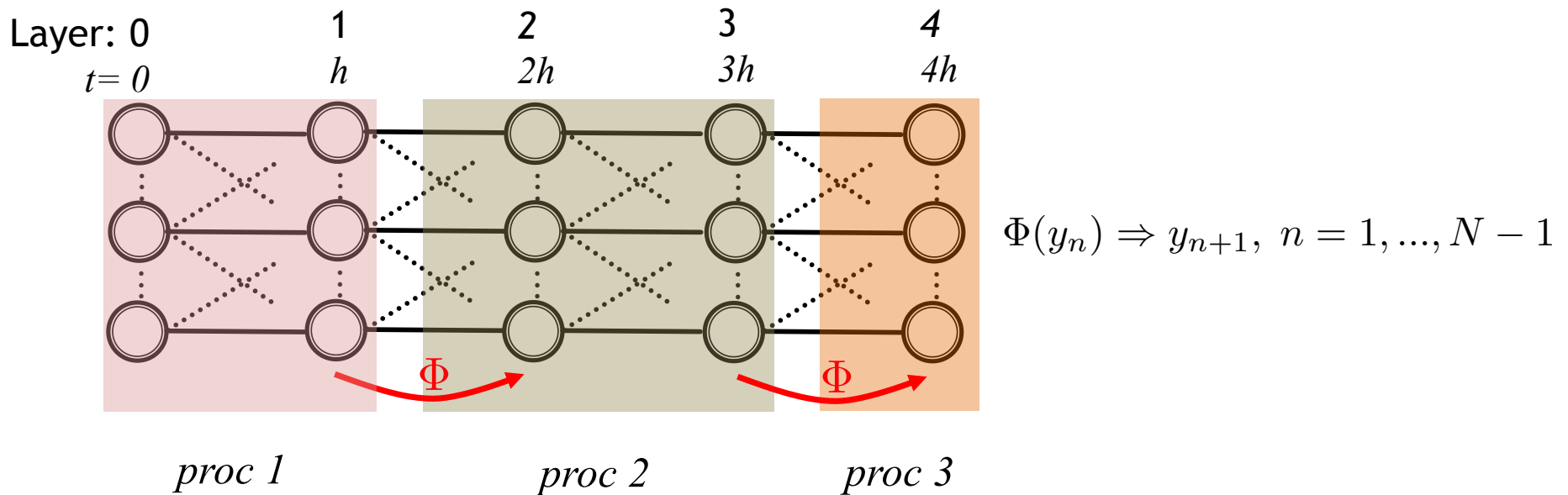
Carry out local layer-to-layer propagation in parallel

Similar to a 1D red-black coloring of the layers

# Parallel-in-time for DNNs

Layer  $i \equiv i$ -th time-step

- Network state  $y_i$  is "like" a spatial vector
- Distribute layers (time-steps) across  $N$  processors with parallel-in-time
- Forwards (backwards) network evaluations done iteratively in parallel
- Time-step  $\Phi$  : Propagation from layer  $k$  to layer  $k+1$



Carry out local layer-to-layer propagation in parallel

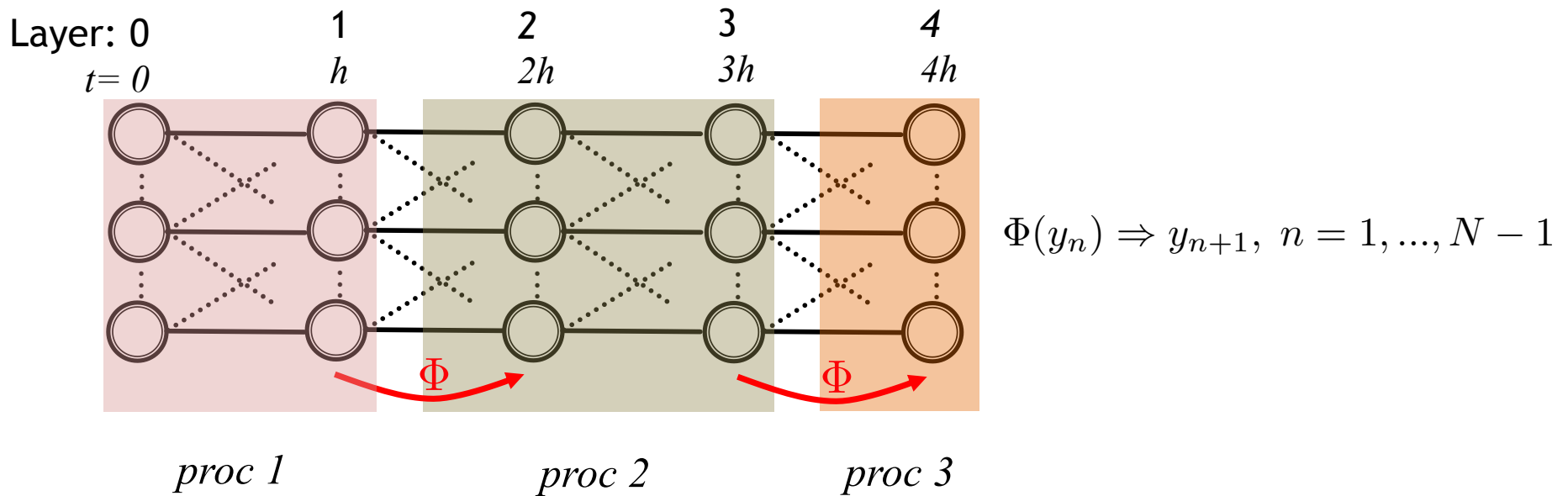
Similar to a 1D red-black coloring of the layers



# Parallel-in-time for DNNs

Layer  $i \equiv i$ -th time-step

- Network state  $y_i$  is "like" a spatial vector
- Distribute layers (time-steps) across  $N$  processors with parallel-in-time
- Forwards (backwards) network evaluations done iteratively in parallel
- Time-step  $\Phi$  : Propagation from layer  $k$  to layer  $k+1$



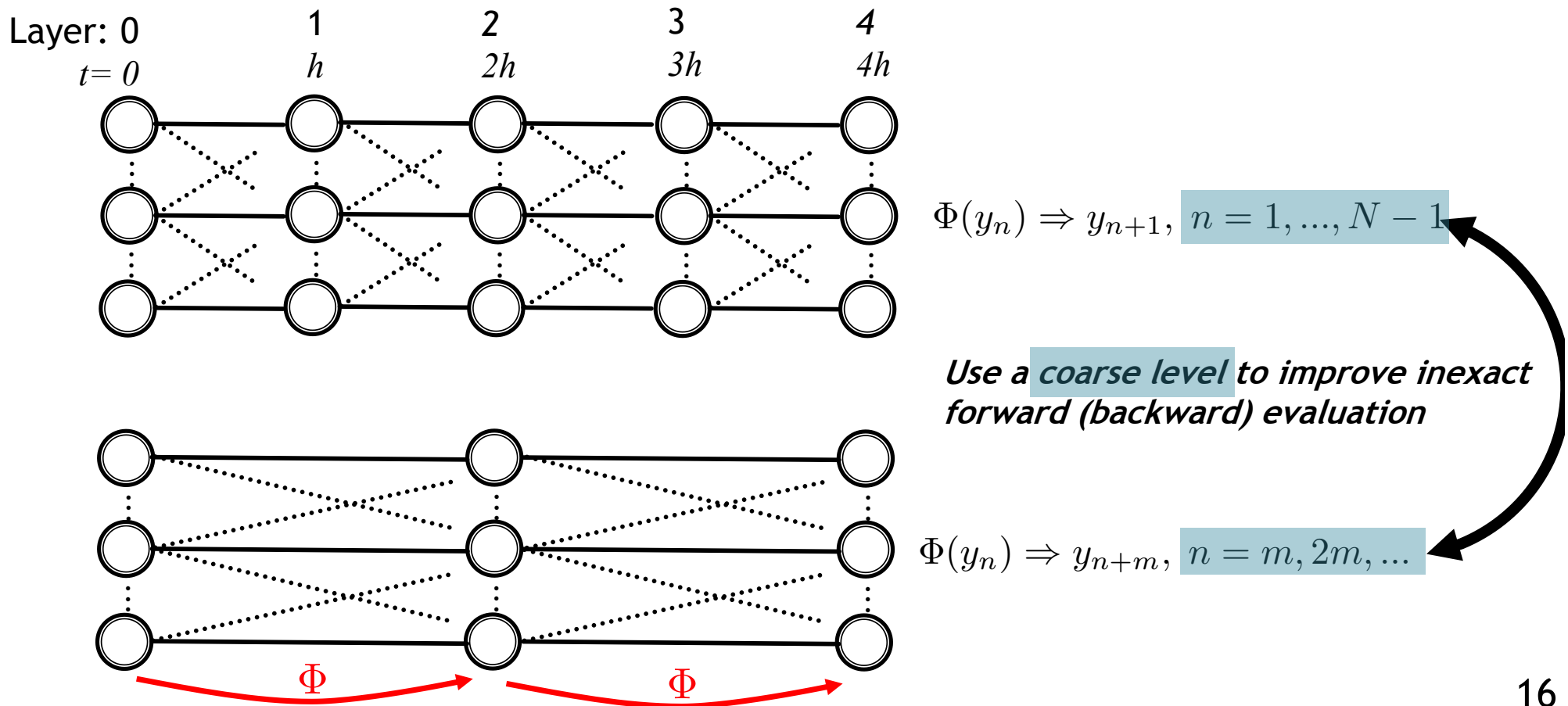
Result is a layer-parallel forward (backward) inexact network evaluation on the finest level

But, how do we account for longer-range interactions? A coarse-grid!

# Parallel-in-time for DNNs

Layer  $i \equiv i$ -th time-step

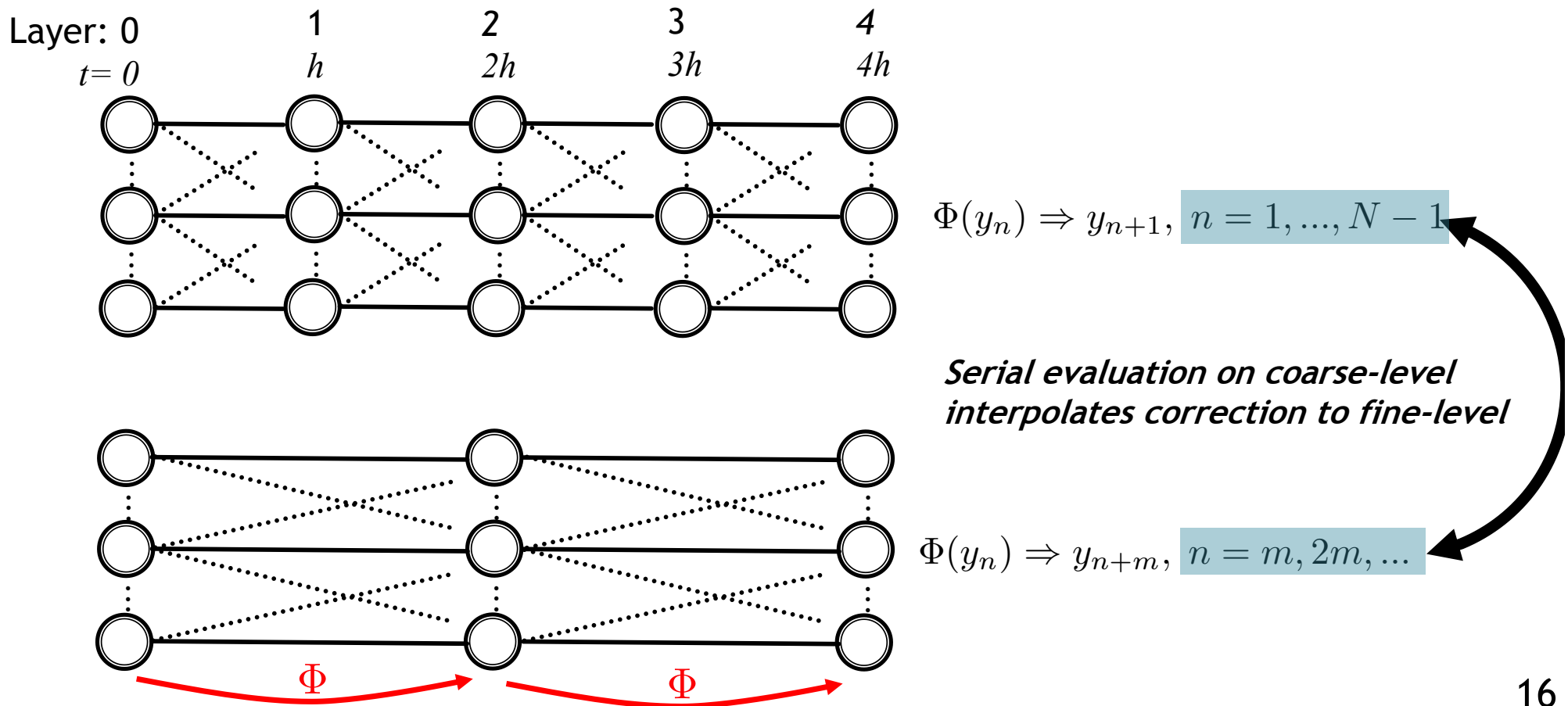
- Network state  $y_i$  is "like" a spatial vector
- Distribute layers (time-steps) across  $N$  processors with parallel-in-time
- Forwards (backwards) network evaluations done iteratively in parallel
- Time-step  $\Phi$ : Propagation from layer  $k$  to layer  $k+1$



# Parallel-in-time for DNNs

Layer  $i \equiv i$ -th time-step

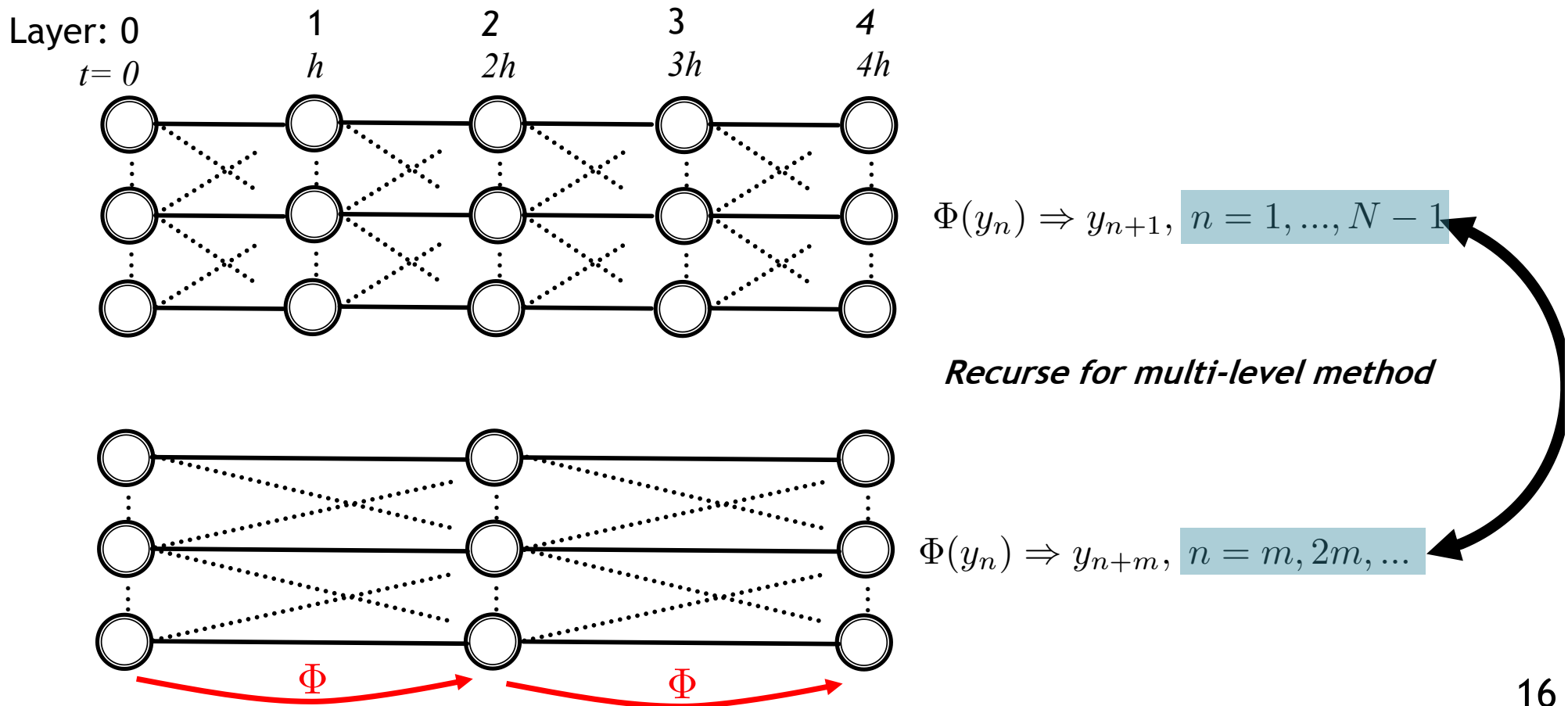
- Network state  $y_i$  is "like" a spatial vector
- Distribute layers (time-steps) across  $N$  processors with parallel-in-time
- Forwards (backwards) network evaluations done iteratively in parallel
- Time-step  $\Phi$  : Propagation from layer  $k$  to layer  $k+1$



# Parallel-in-time for DNNs

Layer  $i \equiv i$ -th time-step

- Network state  $y_i$  is "like" a spatial vector
- Distribute layers (time-steps) across  $N$  processors with parallel-in-time
- Forwards (backwards) network evaluations done iteratively in parallel
- Time-step  $\Phi$  : Propagation from layer  $k$  to layer  $k+1$



# Multigrid-in-Time for an Evolution Problem

- Consider the **general** one-step method

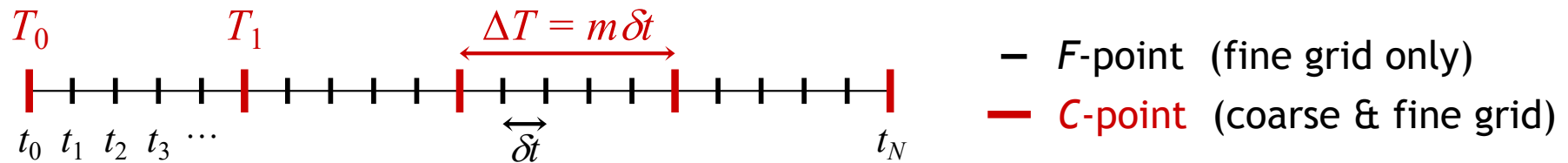
$$\mathbf{u}_i = \Phi_i(\mathbf{u}_{i-1}) + \mathbf{g}_i, \quad i = 1, 2, \dots, N$$

- Below, time marching  $\equiv$  forward solve
  - This is an  $O(N)$  direct method, **but sequential**

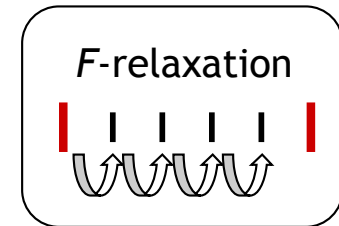
$$A\mathbf{u} \equiv \begin{pmatrix} I & & & & \\ -\Phi & I & & & \\ & \ddots & \ddots & & \\ & & & -\Phi & I \end{pmatrix} \begin{pmatrix} \mathbf{u}_0 \\ \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_N \end{pmatrix} = \begin{pmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_N \end{pmatrix} \equiv \mathbf{g}$$

- We solve this system **iteratively** with multigrid reduction in time (**MGRIT**)
  - Extend multigrid reduction (MGR, 1979) to the time dimension
  - Coarsens only in time  $\rightarrow$  **non-intrusive, i.e.  $\Phi$  is arbitrary, PDE or non-PDE-based**
  - $O(N)$ , highly parallel

# Multigrid reduction in time (MGRIT)



- Relaxation is highly parallel
  - Block-Jacobi alternating between F-points and C-points



- Coarse system is a time rediscrretization with  $N/m$  block rows
  - Approximate impractical  $\Phi^m$  with  $\Phi_\Delta$  a rediscrretization with  $\Delta T$

$$A_\Delta = \begin{pmatrix} I & & & & \\ -\Phi^m & I & & & \\ & \ddots & \ddots & & \\ & & & -\Phi^m & I \end{pmatrix} \Rightarrow B_\Delta = \begin{pmatrix} I & & & & \\ -\Phi_\Delta & I & & & \\ & \ddots & \ddots & & \\ & & & -\Phi_\Delta & I \end{pmatrix}$$

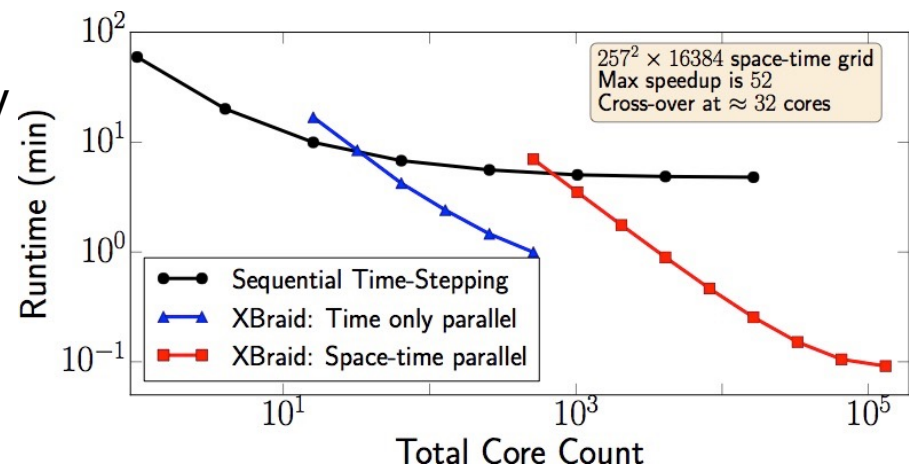
- Apply recursively for multilevel hierarchy

# A broad summary of MGRIT

- Expose concurrency in the "time" dimension with multigrid
- **Non-intrusive**, with unchanged fine-grid problem
- Optimal for variety of parabolic problems
- Converges to **same solution** as sequential marching  
→ Learning problem is essentially unchanged
- Extends to **nonlinear** problems with FAS formulation
- In simple two-level setting, MGRIT  $\equiv$  Parareal

$$\begin{pmatrix} I & & & & \\ -\Phi & I & & & \\ & \ddots & \ddots & & \\ & & & -\Phi & I \end{pmatrix}$$

- Large speedups available, but in a new way
  - Time stepping is already  $O(N)$
  - Useful only beyond a crossover
  - More time steps  $\rightarrow$  more speedup potential
  - *XBraid* is our MGRIT code



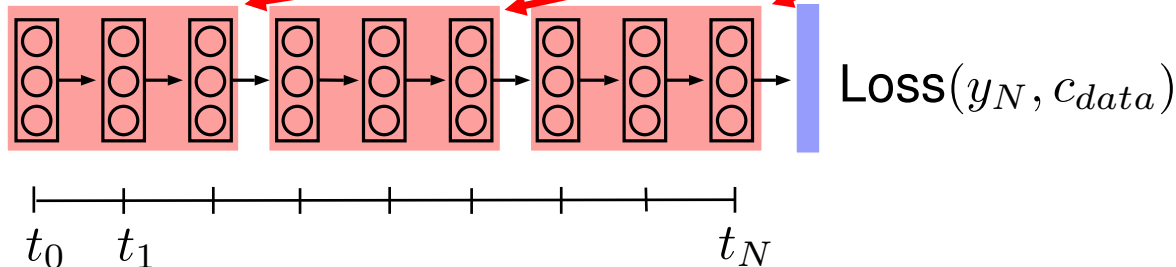
# Parallel-in-time and ODE-like neural networks

- Network propagation is equivalent to a forward Euler discretization, and backpropagation is equivalent to discrete adjoint!
  - Remember:  $\Phi$  is layer-step in a DNN
  - Use equivalence to apply XBraid to forward and backward problems

Assign each block of layers to different procs

- Parallel-in-time goals<sup>1</sup>

- Treat layers as time-steps and apply MGRIT



- Good strong and weak scaling with respect to number of network layers
  - Train a network with 5 layers with same wall-clock time as 1000 layers
- Solve the same training problem (no shortcuts) as the sequential training version
- Provide novel layer-parallelism (decoupled layer computations in parallel)

20



# Outline

- Need for parallelism driven by DNNs
  - Consider various forms of parallelism
  - Leverage wealth of parallel algorithms available from the CS&E community
- Deep neural networks
  - Interpret as an ODE (ResNets, Transformers, GRU)
- Layer-parallelism
  - Parallel-in-time applied to layers-as-time-steps
- Results
  - Code (TorchBraid)
  - Combine with coarsening inside layer (spatial coarsening)
  - Combine with data-parallelism
  - Tiny ImageNet: Combine with pooling and batch-norm
  - Transformers

# Code: Layer-parallelism with multigrid



- **TorchBraid: XBraid (MGRIT code in MPI/C) and PyTorch**

<https://github.com/Multilevel-NN/torchbraid>

```
$ pip install path/to/torchbraid
```

- Python, C/MPI coupling with Cython (a little messy, but also impressive)
- Parallel Dataset and Dataloader functions inherited from PyTorch
  - Only load on root processor
- GPU-to-GPU direct communication key for performance
  - Requires CUDA-capable MPI, which you can test with

```
$ make tests -direct -gpu
```

# Code: Layer-parallelism with multigrid



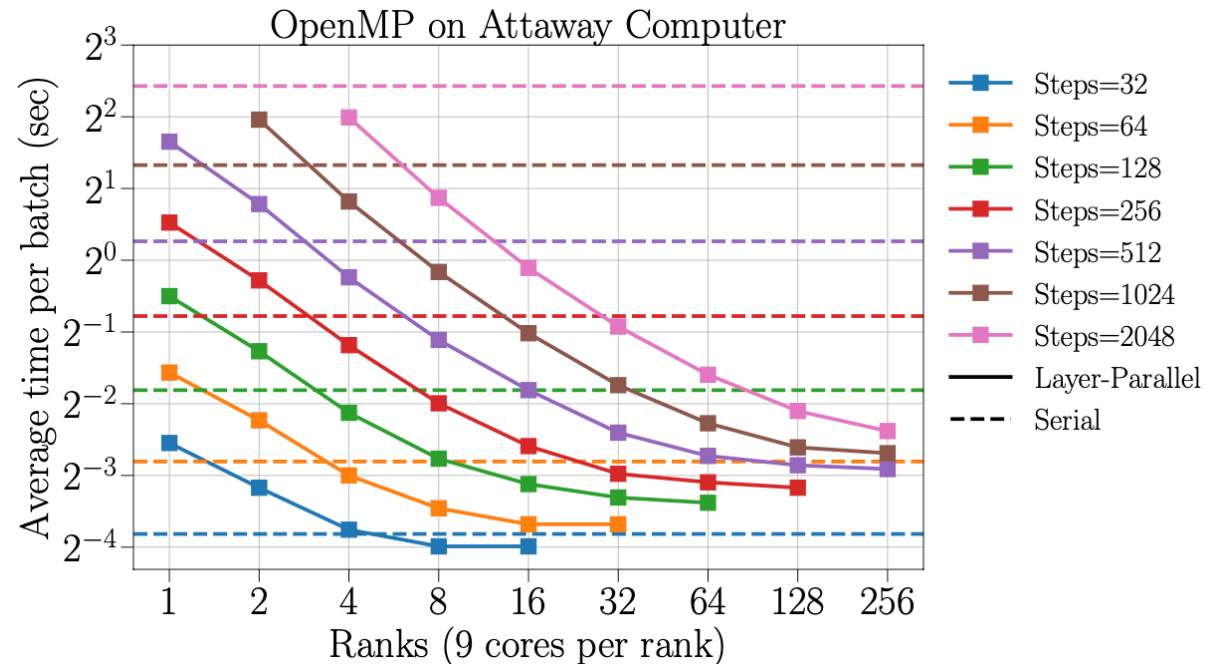
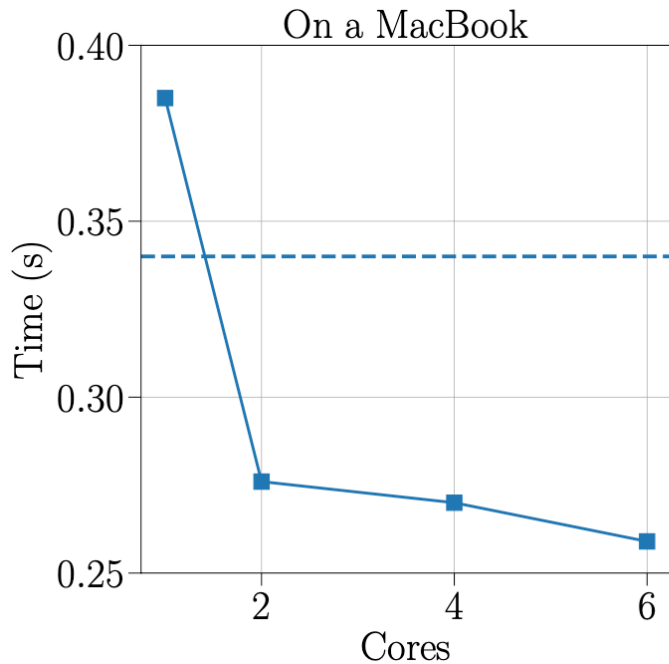
- Automatic differentiation (AD) done over MPI with distributed layers
  - Designed so root rank behaves like serial training when only one processor is used
  - During last MGRIT iteration, AD is turned on and constructs a correct computational graph in the distributed setting
    - Backward inherits from PyTorch and sends gradient info between ranks with MPI

## Sample TorchBraid Code:

	Rank 0	Rank 1
1. <code>optimizer.zero_grad()</code>	zero grads.	zero grads.
2. <code>output=model(data)</code>	MGRIT fwd. prop., recv. last layer	MGRIT fwd. prop., send last layer
3. <code>loss=compose(criterion, output, target)</code>	evaluate loss	no-op
4. <code>loss.backward()</code>	MGRIT bwd. prop	MGRIT bwd. prop.
5. <code>optimizer.step()</code>	update params.	update params.

# Results: Simple MNIST

- To begin, consider an ODENet (no pooling or layer-norm) and MNIST
- Simplest example using only CPUs; Dense closing layer w/ softmax

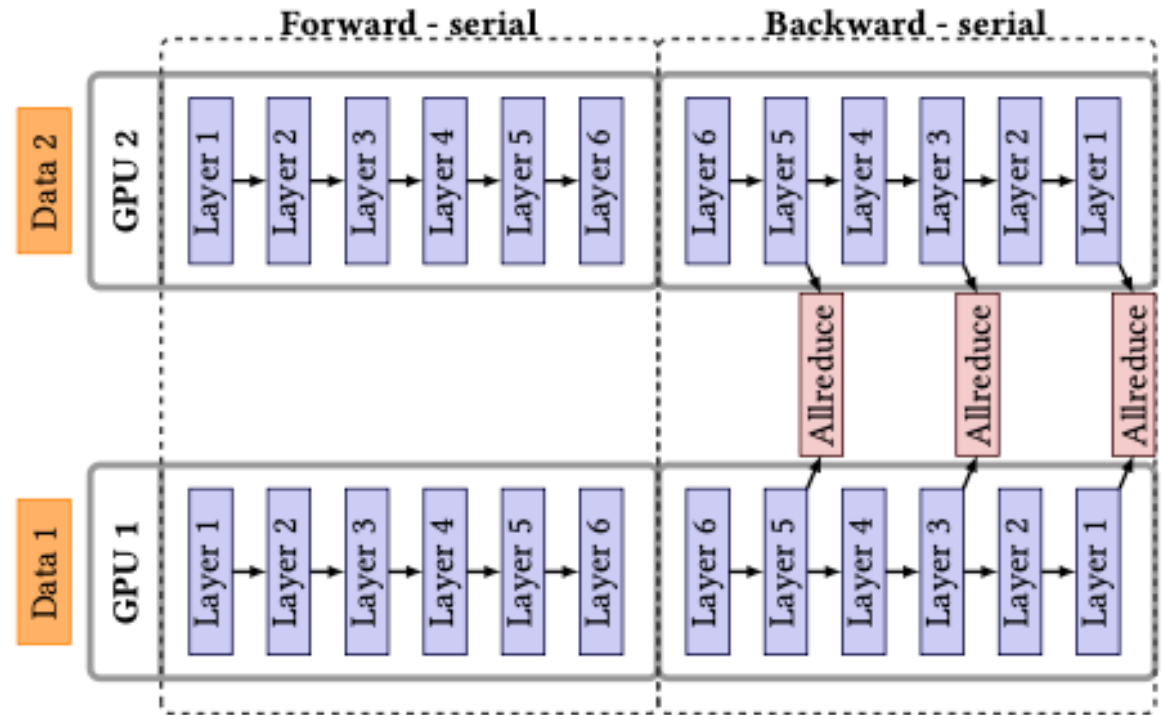


- Speedup up at 2 cores on laptop (36 steps, 3 channels)
- Larger networks and larger computers → large speedup possibilities, 10x–20x, ...

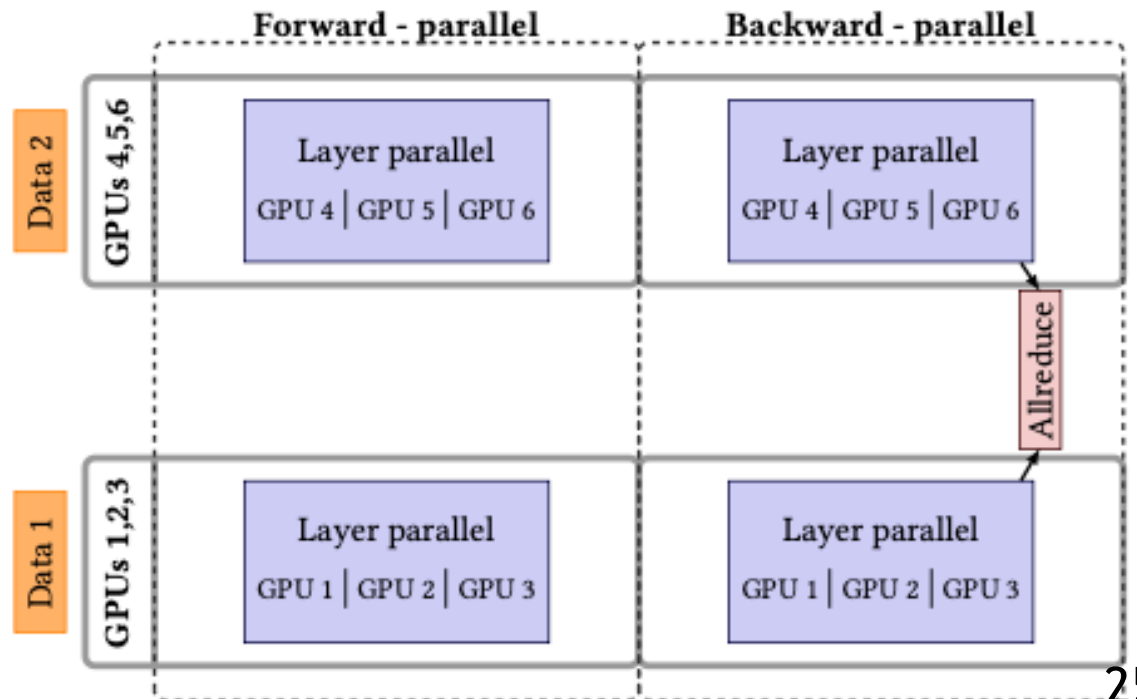
# Results: Layer-parallelism and data-parallelism

- Data-parallelism proceeds as normal
  - Split the MPI communicator into sub-communicators for layer- and data-parallelism
- ➔ More parallelism!

Data-Parallelism

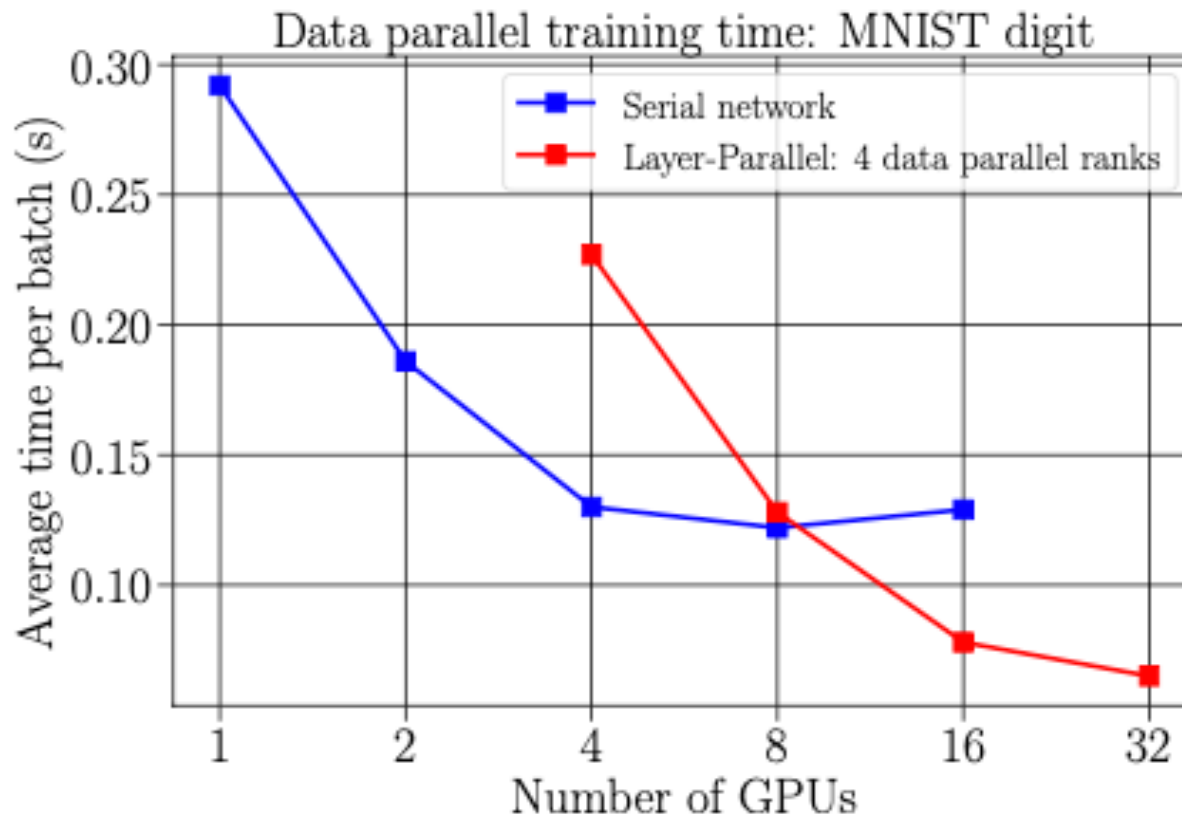


Layer- + Data-Parallelism



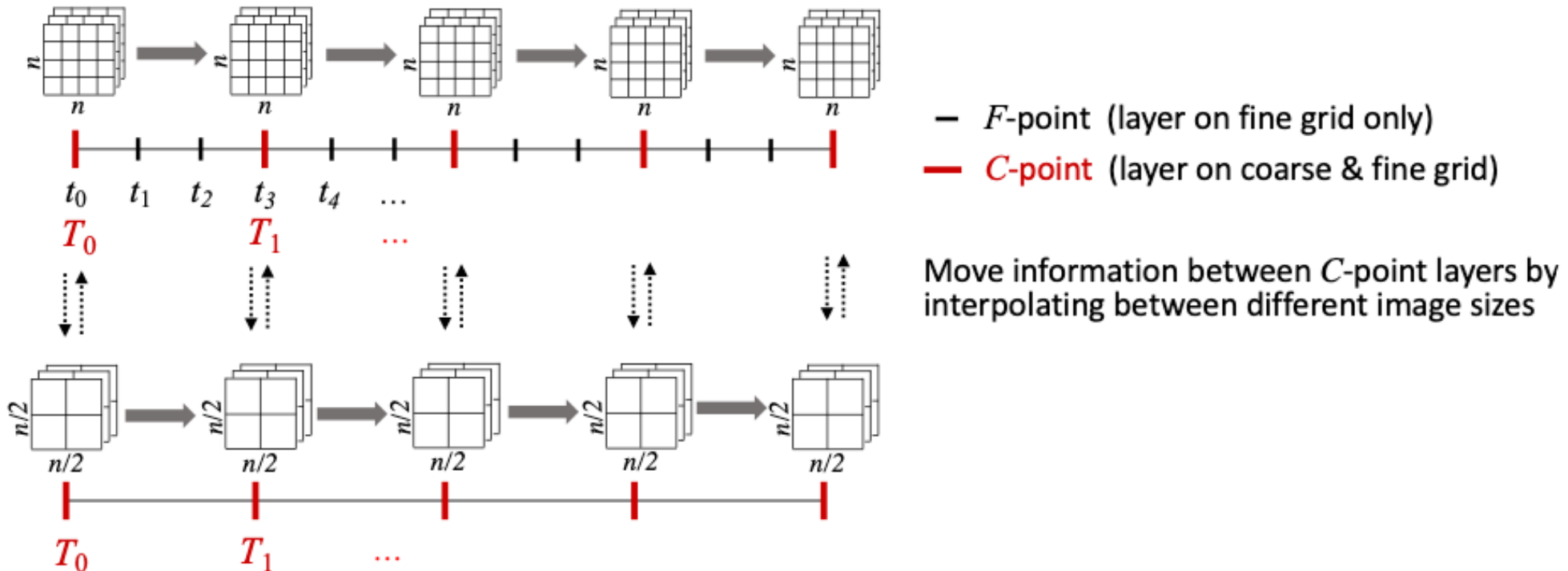
# Results: Layer-parallelism and data-parallelism

- For demonstration purposes: 12 channels and 128 steps used on MNIST
- Serial network uses only data parallelism, which quickly exhausts
- Layer-parallelism allows for extra dimension of parallelism and speedup
  - Easy to use sub-package: `torchbraid.utils.data_parallel`



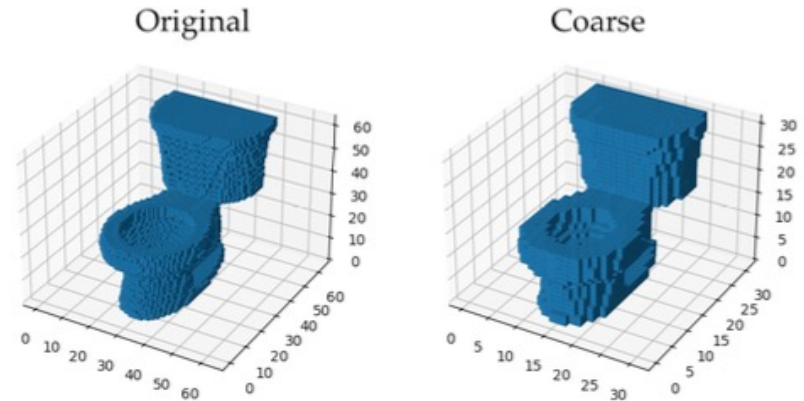
# Results: Coarsening inside layers

- Coarsen in space (inside layers) for extra performance
- Particularly efficient for 3D problems
  - Coarse MGRIT levels decrease in "space" by a factor of 8
  - Distribute large dataset and large network (which could be too large for a single GPU) across many GPUs
- Sample MGRIT hierarchy

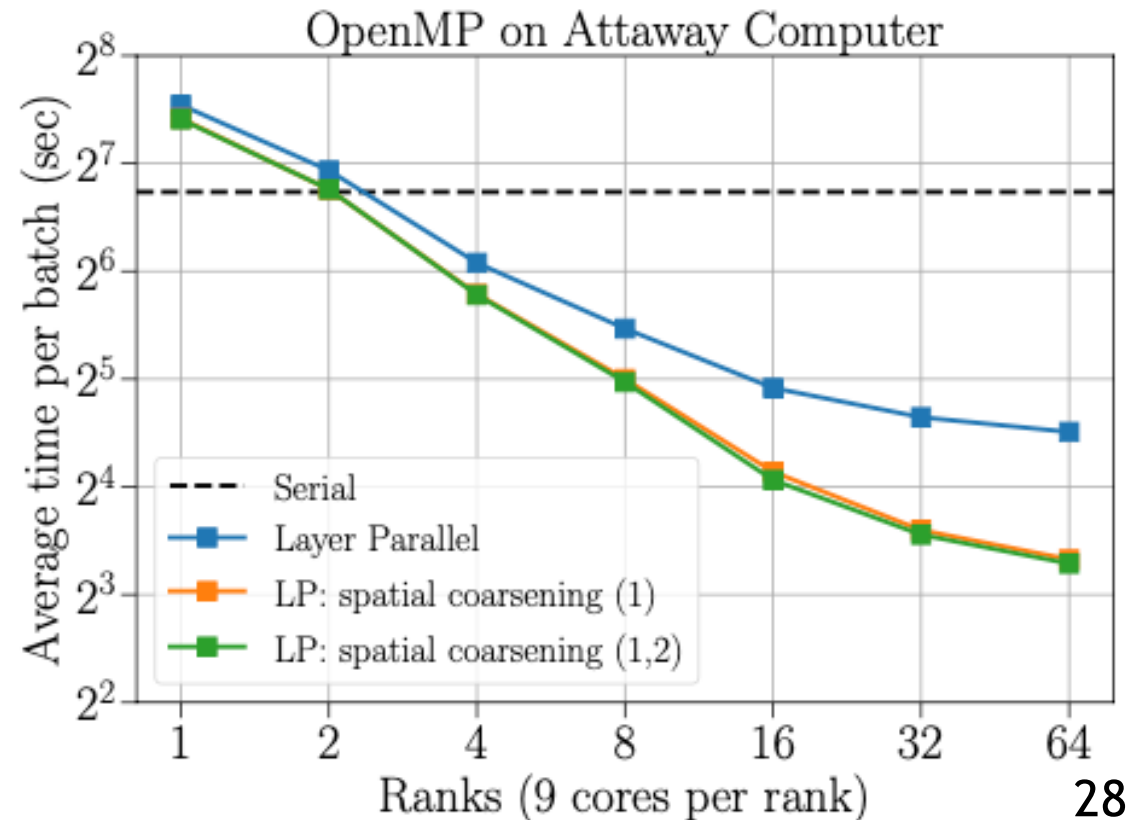


# Results: Coarsening inside layers

- ModelNet10 dataset of 3D object recognition of various furniture items
  - $64^3$  sized images
  - 4K training images, 1K validation images



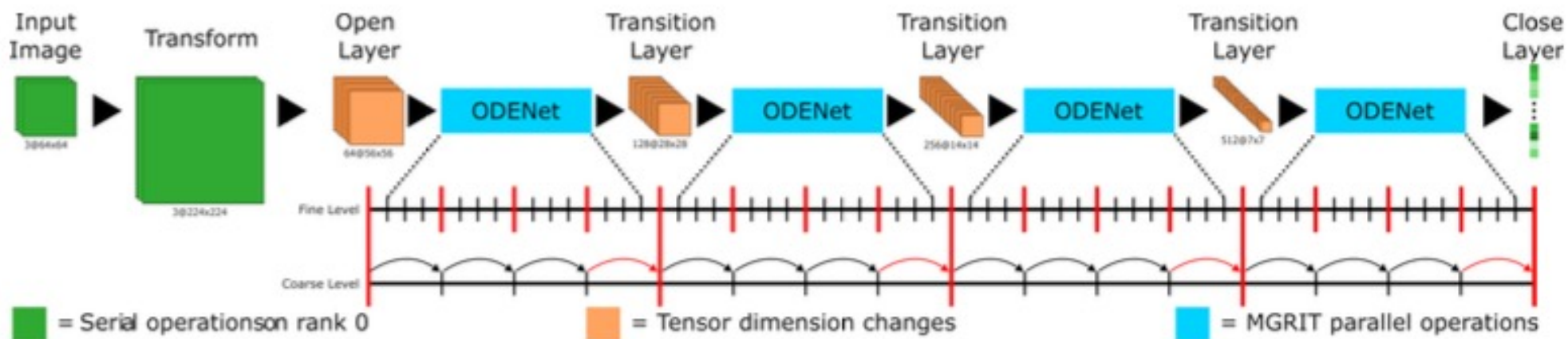
- 128 layers, 3D convolutional ResNet with ReLU
- Opening: expand to 4 channels
- Dense closing layer w/ softmax
- Coarsening in layer provides substantial speedup!
  - 2.3x over layer-parallel
  - 11.0x over serial
  - Option (1) best accuracy





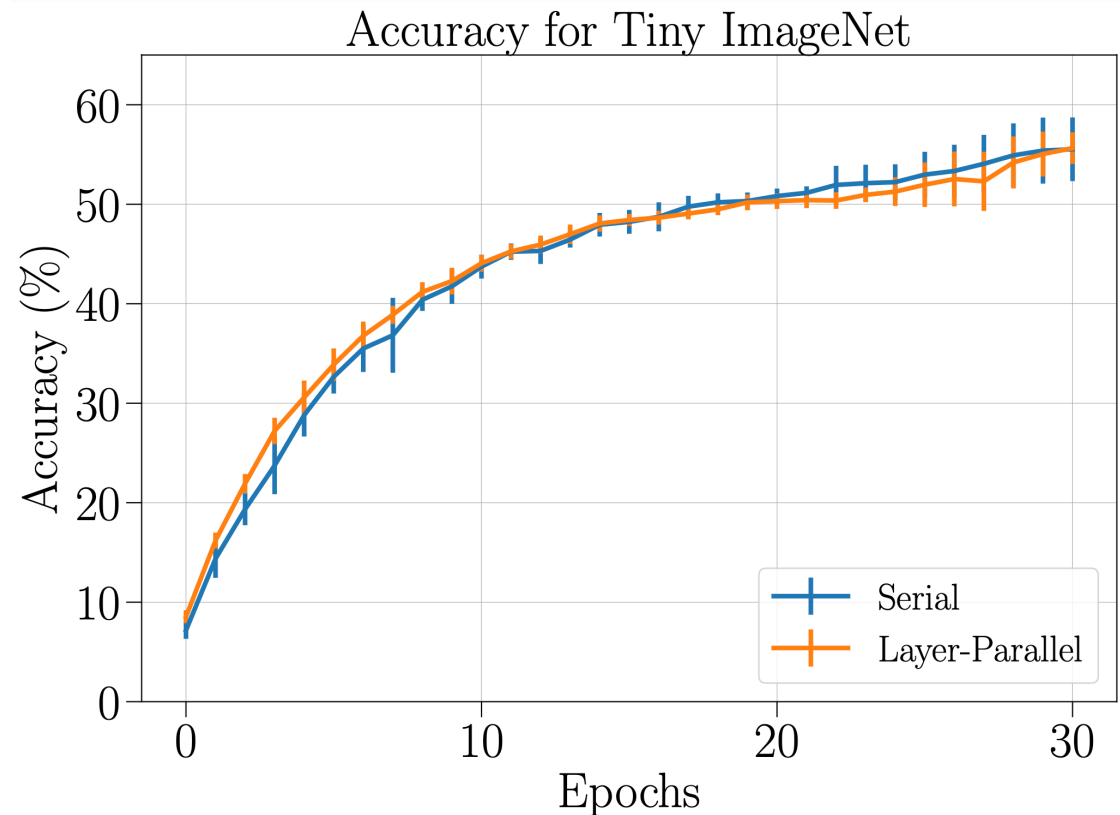
# Results: Tiny ImageNet

- Extend TorchBraid ODEnets to use max pooling and batch normalization
- Maintain max pooling layers as C-points on coarse levels
- Layer-parallel batch normalization
  - During inference, batch norm is identical to standard
  - During training, batch norm running averages are only updated during final fine-level evaluation at a layer



# Results: Tiny ImageNet

- Train on PLEIADES cluster at University of Wuppertal
  - 5 compute nodes, each containing 8 A100 GPUs
  - Utilize GPUs for training and compare to serial time on GPU
- Training setup
  - 256 total layers
  - 64 layers per ODENet block
  - 30 epochs and batch size of 50
  - 16 training runs
    - Vertical bars: std. dev
- Accuracy essentially unchanged



# Results: Tiny ImageNet

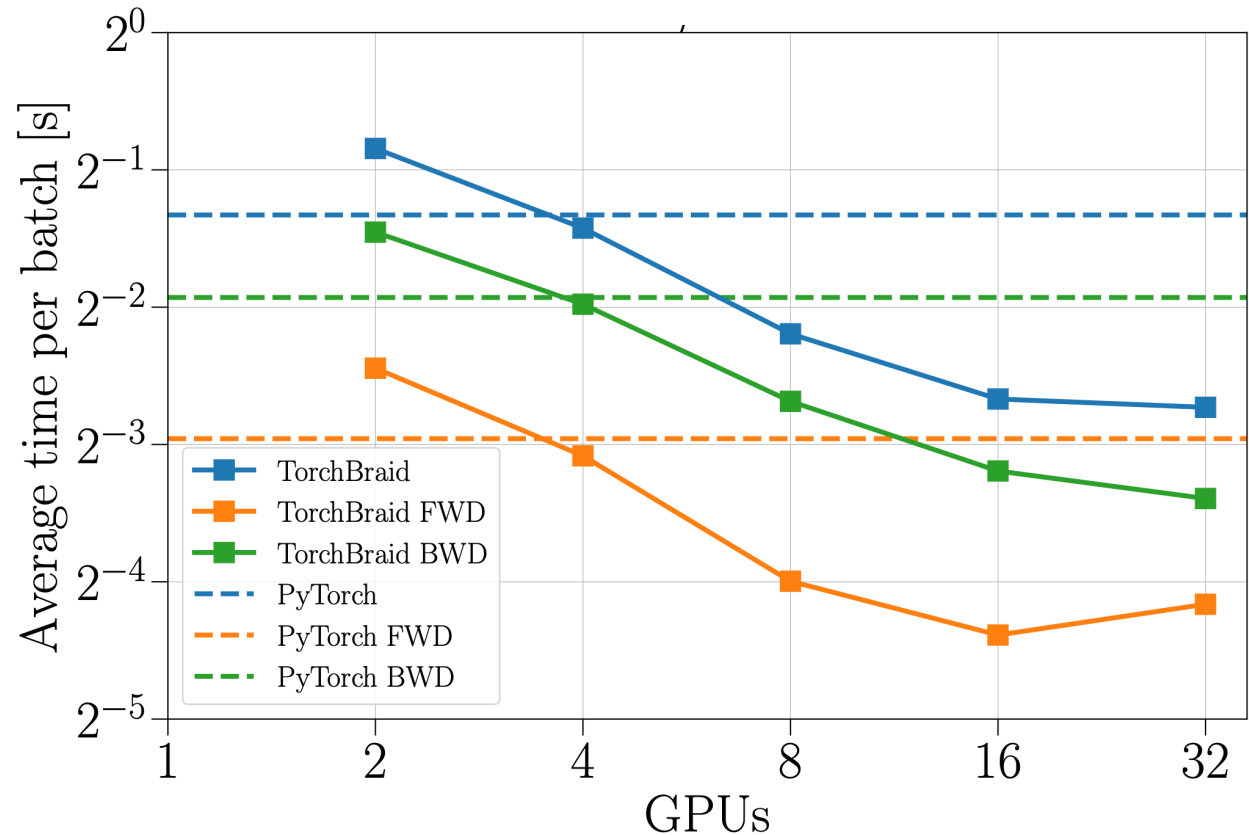
- Strong scaling on Glinda

- One A100 per node
- Compare forward, backward, and overall times

- Crossover point: 4 GPUs

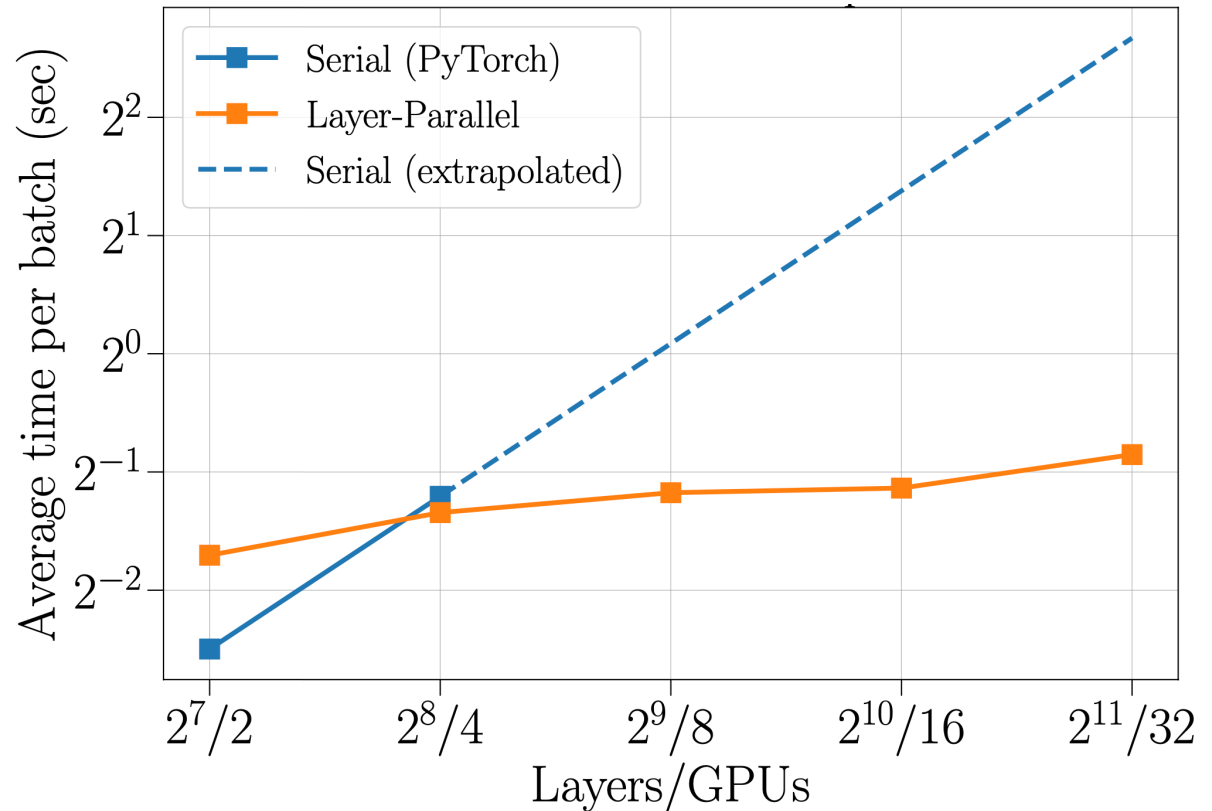
- 2.5x speedup at 16 GPUs

- Major coding effort! But, still work to be done on efficiency!



# Results: Tiny ImageNet

- Weak scaling on Glinda
  - One A100 per node
  - Work per GPU fixed, and consider larger networks
- Serial has two data points due to out of memory
  - Extrapolate linearly
- Layer-Parallel: scales well
  - Large speedup potential for large networks
  - Distributed network allows for much more memory!



# Results: Layer-parallelism and Transformers

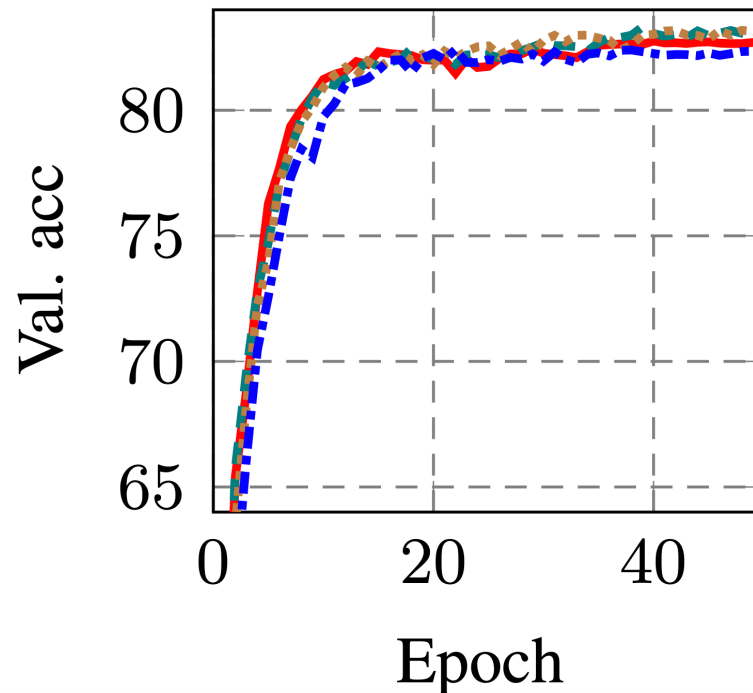
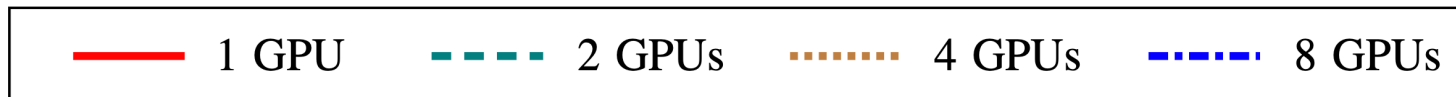
- Apply layer-parallel to ODE-inspired transformer
  - Piz-Daint, Swiss National Supercomputer Center, 1 Tesla P100 GPU per node
  - Leading the effort: student, Marc Salvado, and post-doc, Marshall Jiang
- Morphological classification (noun, verb, adverb, ...) with GUM corpus<sup>1,2</sup>
- Results shown for encoder only
- Decoder is still a work-in-progress
  - MGRIT convergence rate deteriorates for larger networks
  - Exploring regularization techniques to encourage "continuity" between layers (no sacrifice of learning)
  - Exploring splitting CA and MLP blocks into separate layers for more parallelism

1. *The GUM corpus: Creating multilayer resources in the classroom*, A. Zeldes, Language Resources and Evaluation, 2017.

2. *Universal Dependencies*, J. Nivre, D. Zeman, F. Ginter, and F. Tyers, Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics, 2017.

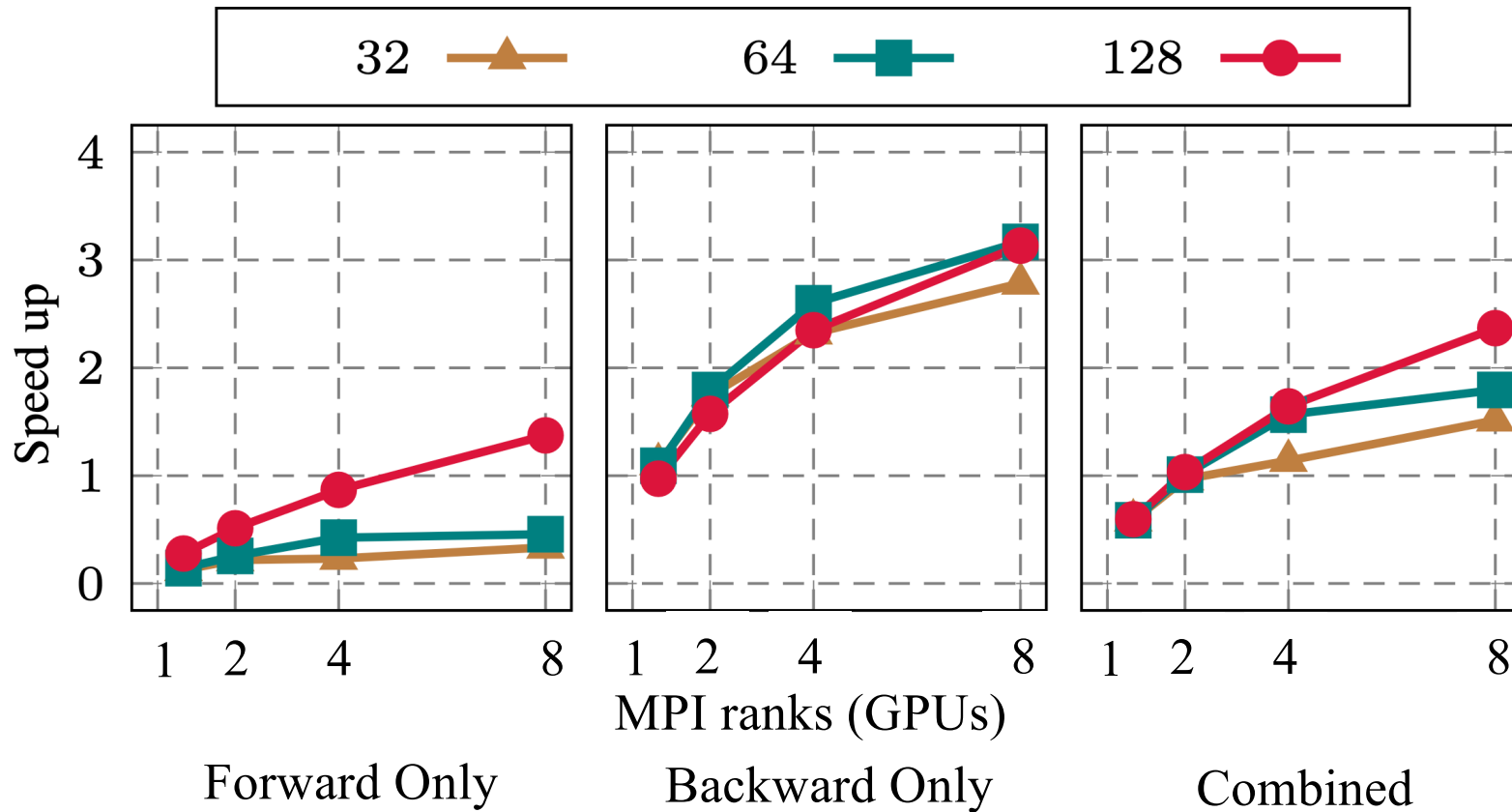
# Results: Layer-parallelism and Transformers

- Morphological classification example with 64 transformer blocks
- 1 GPU represents serial training with PyTorch
- No degradation in learning when using layer-parallel



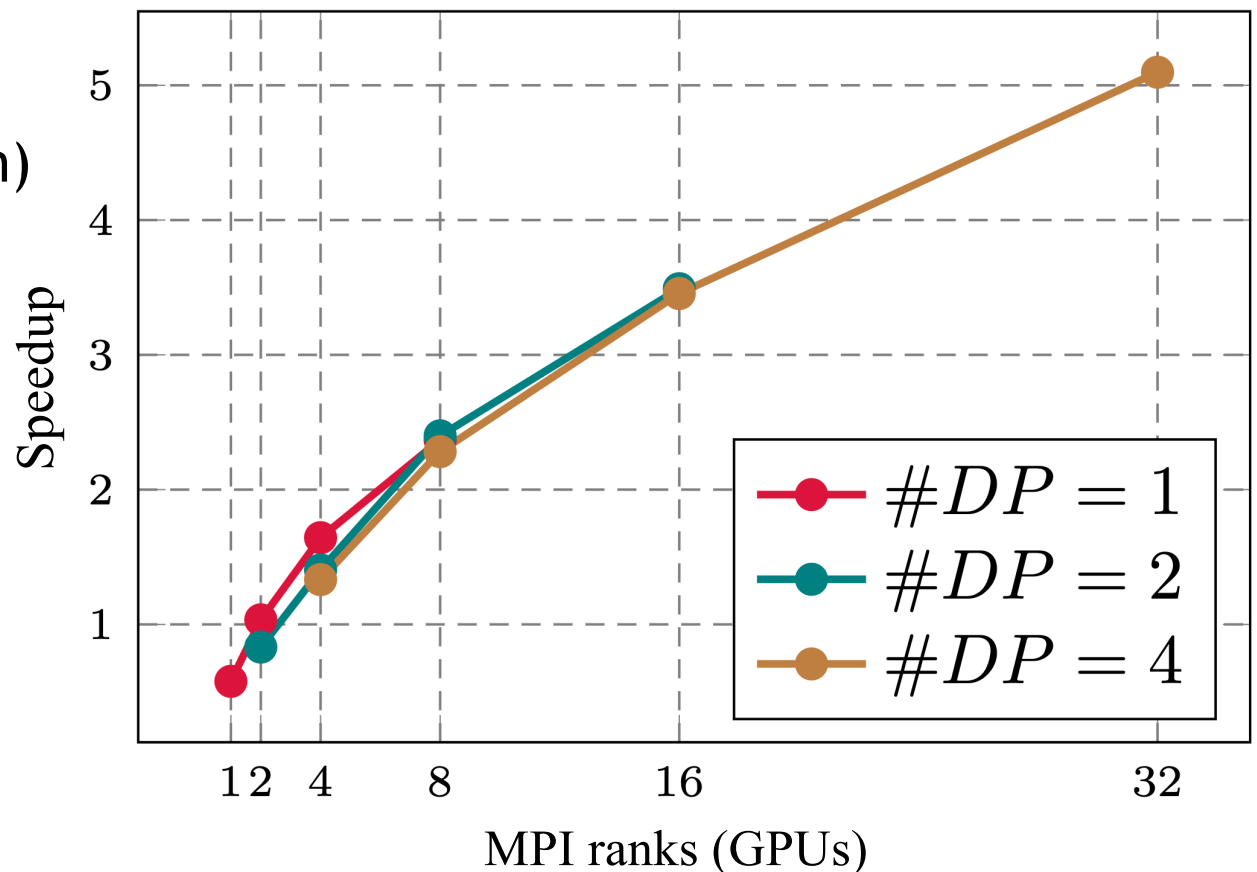
# Results: Layer-parallelism and Transformers

- Maximum speedup at 128 transformer blocks is 2.5x at 8 GPUS
- Baseline is serial PyTorch (no TorchBraid)



# Results: Layer-parallelism and Transformers

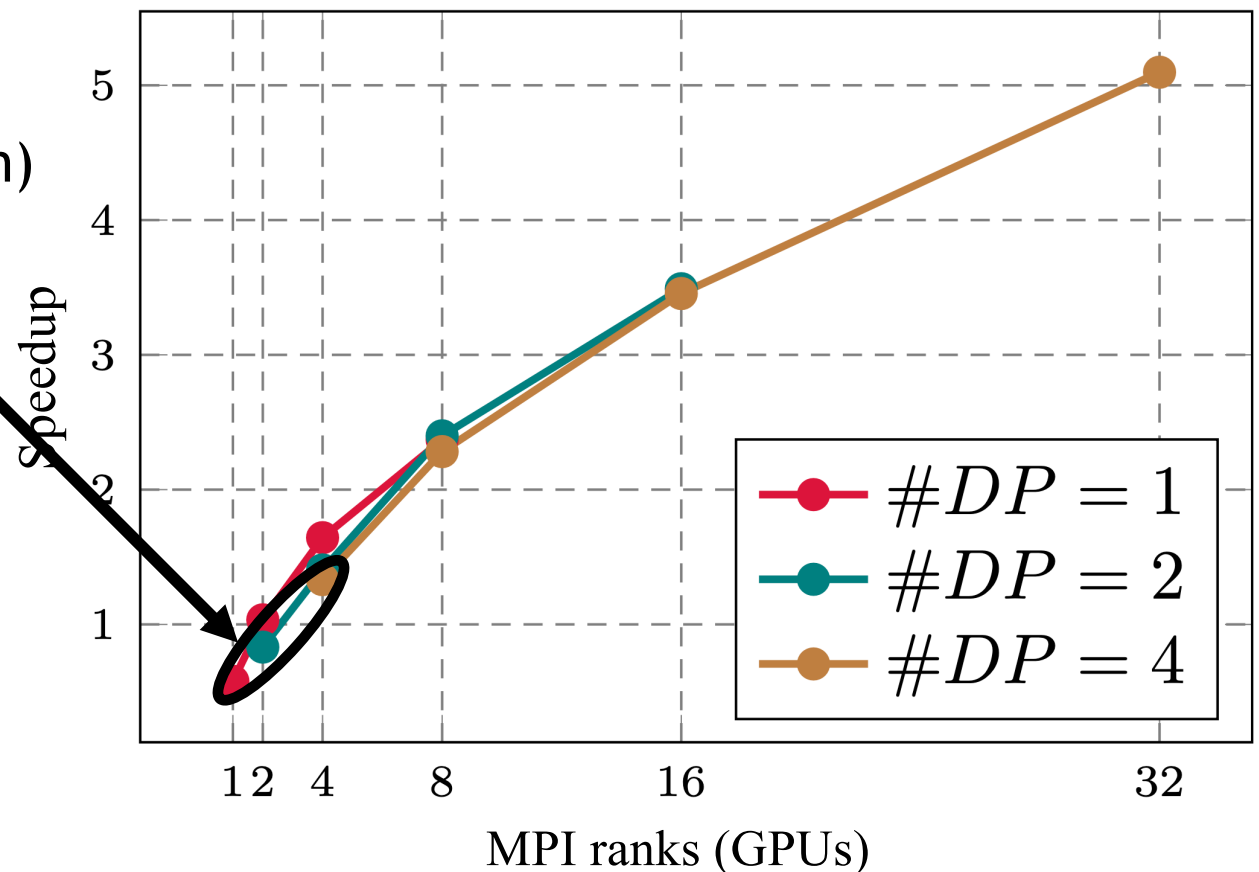
- Greater speedups available when combined with data-parallelism
- Baseline is serial PyTorch (no TorchBraid)
- Data-parallelism with 1, 2, and 4 GPUs (1 GPU in LP dimension)





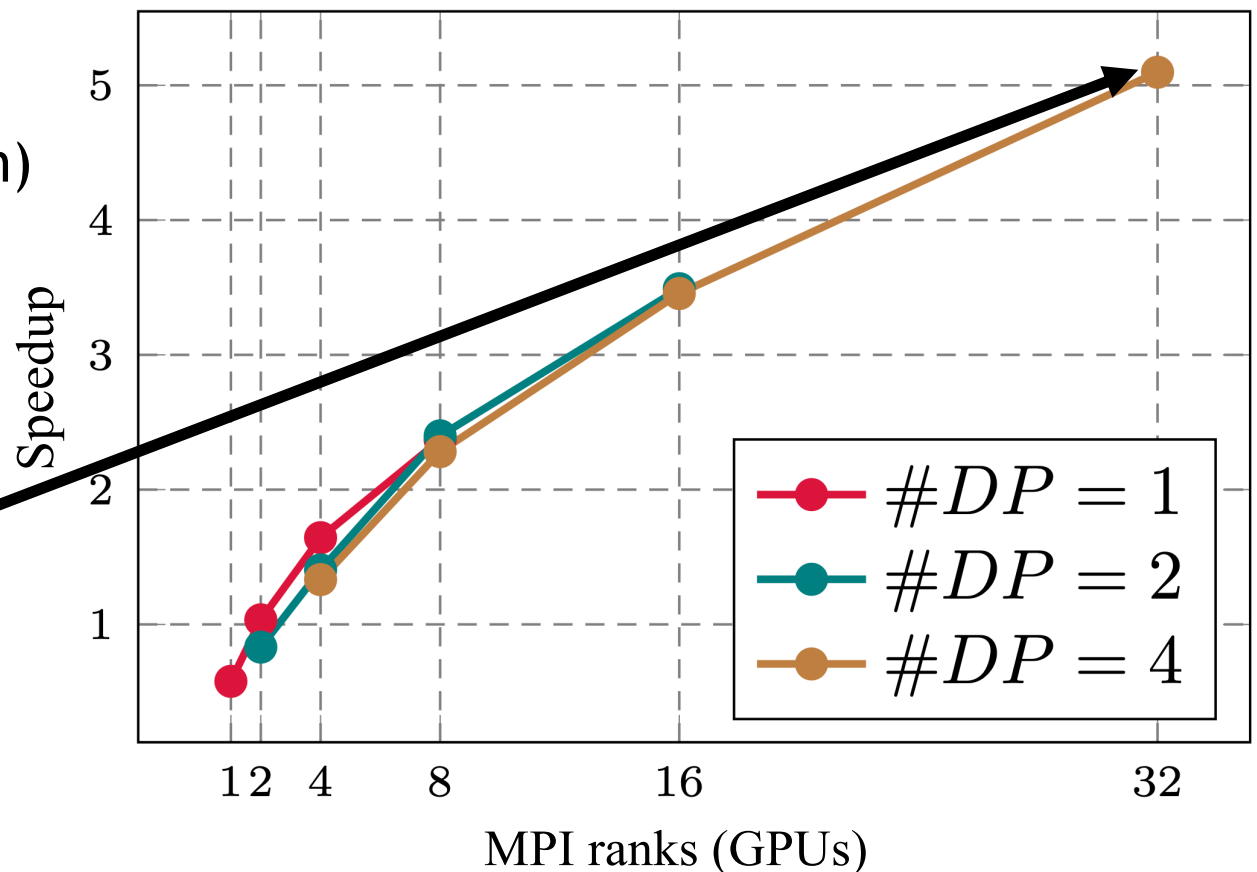
# Results: Layer-parallelism and Transformers

- Greater speedups available when combined with data-parallelism
- Baseline is serial PyTorch (no TorchBraid)
- Data-parallelism with 1, 2, and 4 GPUs (1 GPU in LP dimension)



# Results: Layer-parallelism and Transformers

- Greater speedups available when combined with data-parallelism
- Baseline is serial PyTorch (no TorchBraid)
- Data-parallelism with 1, 2, and 4 GPUs (1 GPU in LP dimension)
- Then, addition of GPUs for LP yields ~5x speedups



# Conclusion: Layer-parallelism allows for parallel speedup across layers

- Applicable to various architectures beyond original ODENet, e.g., ResNets (layer-norm, max-pooling), transformer, GRU
- Finds same gradient as sequential propagation (to within tolerance)
- Combine with data-parallelism to maximize machine utilization and speedup
- Combine with coarsening inside layer for increased speedup and efficiency
- Scales strongly and weakly
- Deeper networks → More speedup
- Distribute large datasets and large networks across MPI
  - Allows for much larger datasets and networks (3D, Videos, ...)
- Philosophically, leverage parallel frameworks from CS&E to accelerate DNNs

## Layer-parallel References

1. Cyr, Hahne, Moore, Schroder, Southworth, Vargas. *TorchBraid: High-Performance Layer-Parallel Training of Deep Neural Networks with MPI and GPU Acceleration*. Submitted.
2. Günther, Ruthotto, Schroder, et al. *Parallel-in-Layer Optimization for Training of Deep Residual Networks*. SIMODS, 2020.
3. Cyr, Günther, Schroder, *Multilevel Initialization for Layer-Parallel DNN Training*. Int. J. Comput. Vis. Sci. Eng. (2021).

## Code

- TorchBraid: layer-parallelism with PyTorch  
<https://github.com/Multilevel-NN/torchbraid>
- XBraid: MGRIT library for layer-parallelism  
<https://github.com/XBraid/xbraid>



## Other References

1. Haber, Ruthotto. *Stable Architectures for Deep Neural Networks*. Inverse Probl., 2017.
2. Falgout, Friedhoff, Kolev, MacLachlan, Schroder, *Parallel Time Integration with Multigrid*, SISC, 2014.
3. Günther, Gauger, Schroder, *Non-Intrusive Parallel-in-Time Adjoint Solver with the XBraid Library*, CVS, 2017.