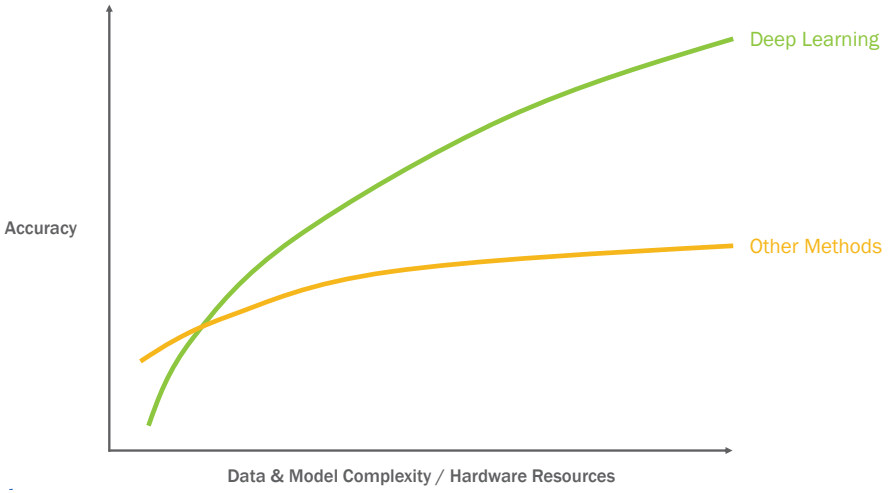
Berkeley
UNIVERSITY OF CALIFORNIA

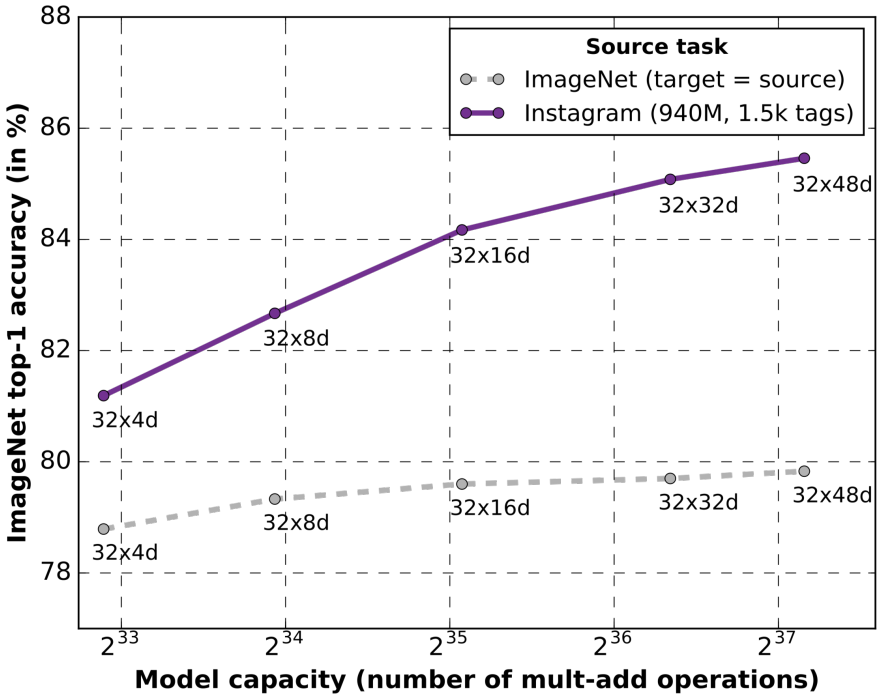
Parallelism in Deep Neural Network Training, with an emphasis on Graph Neural Networks

Aydın Buluç
Computational Research Division, LBNL
EECS Department, UC Berkeley

CS Summer Student Program
June 10, 2021

Modern machine learning is high-performance computing



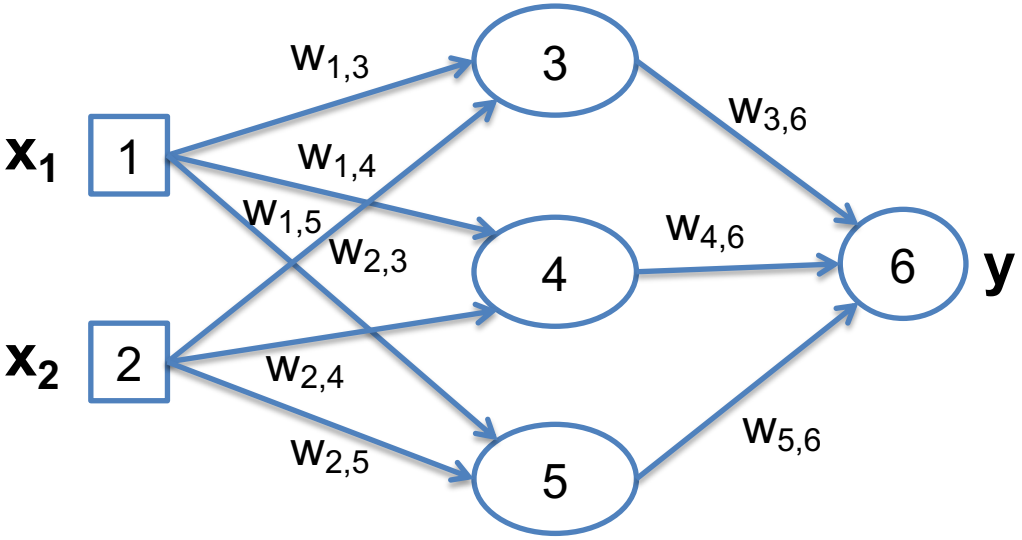


Source: Advancing state-of-the-art image recognition with deep learning on hashtags

Slide source: Misha Smelyanskiy (Facebook AI co-design director)

Training Neural Networks

- Training is to **adjust the weights (W)** in the connections of the neural network, in order to change the function it represents.

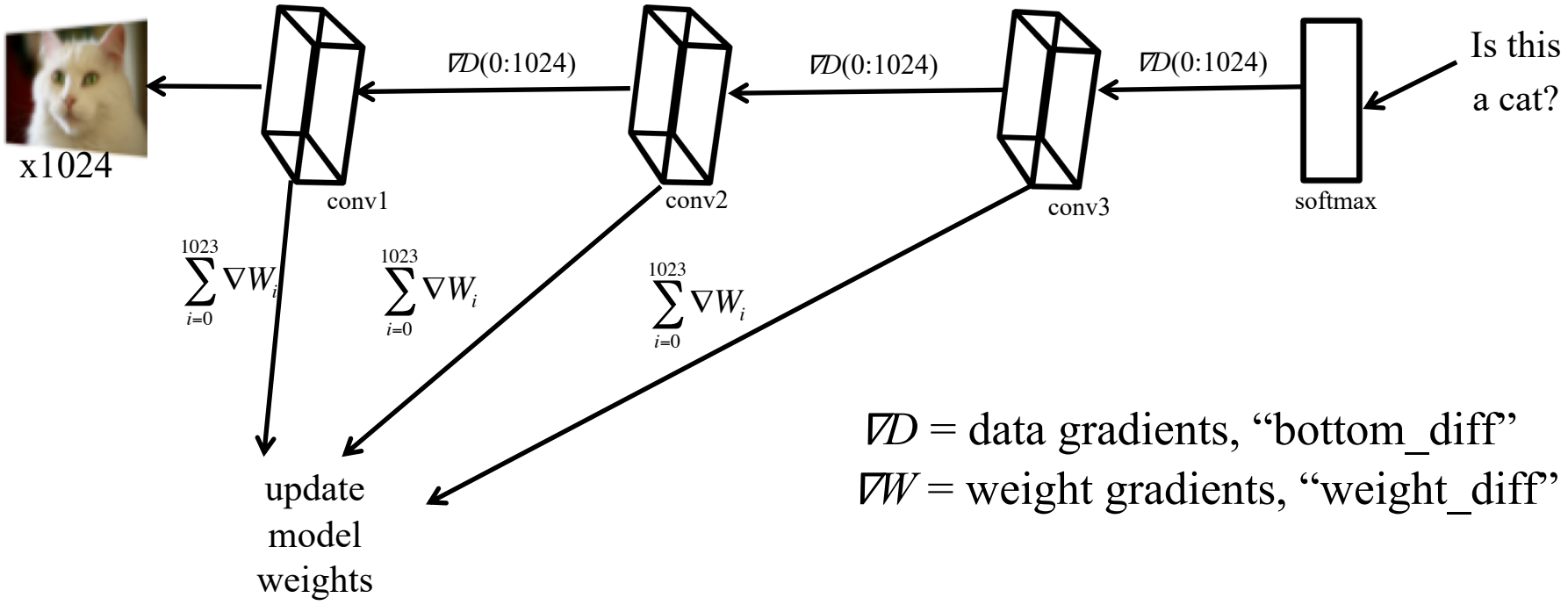


Only parameters are weights for simplicity (i.e. ignore bias parameters)

W: the matrix of weights

A "shallow" neural network with only one hidden layer (nodes 3,4,5), two inputs and one output.

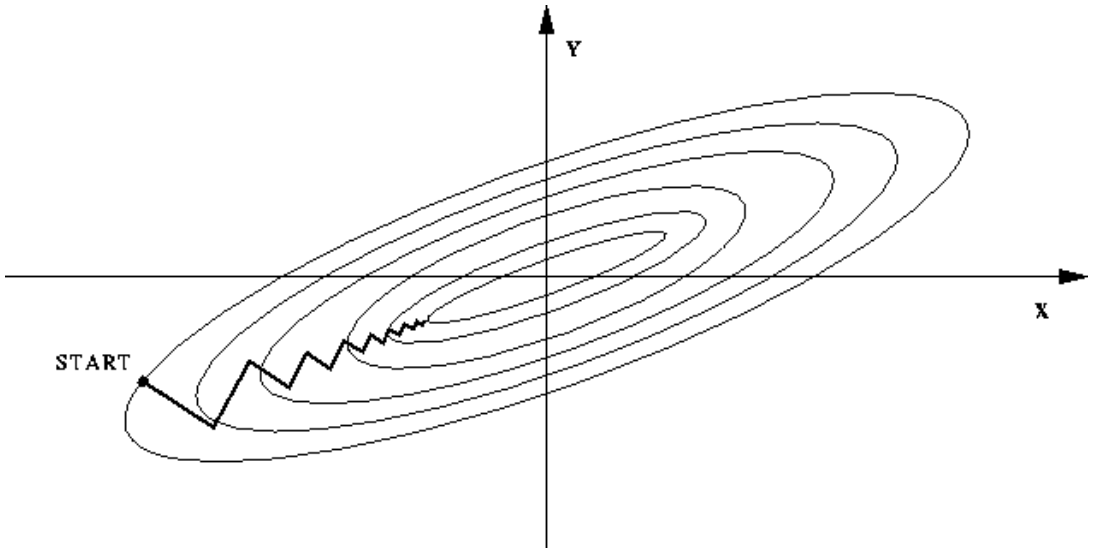
Deep Neural Network Training

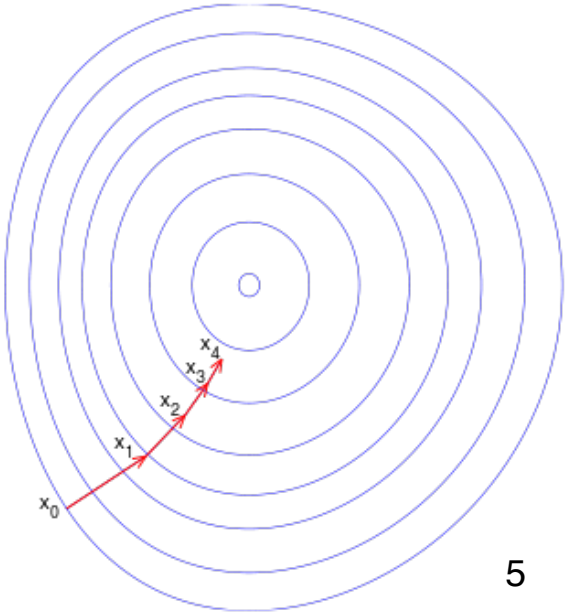


Gradient Descent

$$W^{t+1} \leftarrow W^{t} - \alpha \cdot \nabla_{W} f(W^{t}, x)$$

- Also called the steepest descent algorithm
- In order to minimize a function, move towards the opposite direction of the gradient at a rate of α.
- α is the step size (also called the learning rate)
- Used as the ***optimization backend*** of many other machine learning methods (example: NMF)





Stochastic Gradient Descent (SGD)

$$\text{Assume } f(W^t, x) = \frac{1}{n}\sum_{i=1}^{n} f_i(W^t, x)$$

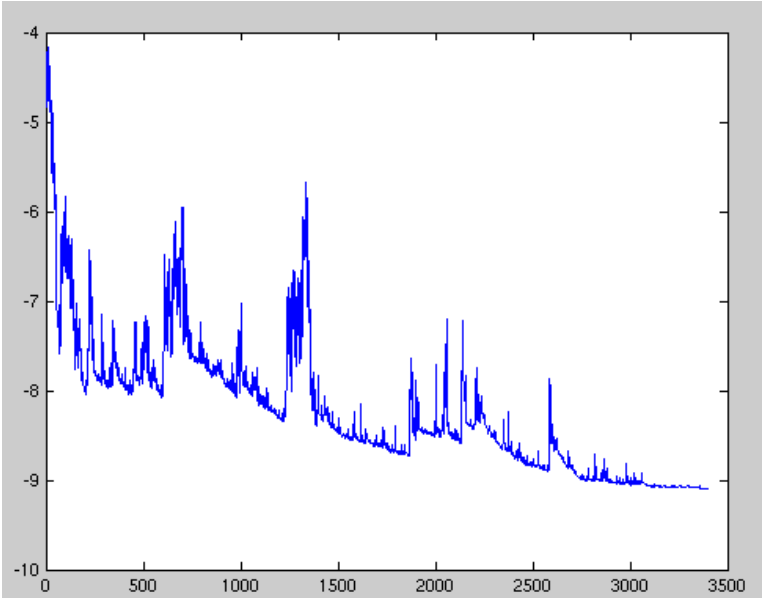
$$W^{t+1} \leftarrow W^t - \alpha \cdot \nabla_W f_i(W^t, x)$$

Pure SGD: compute gradient using 1 sample

$$W^{t+1} \leftarrow W^t - \alpha \cdot \frac{1}{b}\sum_{i=k+1}^{k+b} \nabla_W f_i(W^t, x)$$

Mini-batch: compute gradient using b samples

f is not going down for every iteration

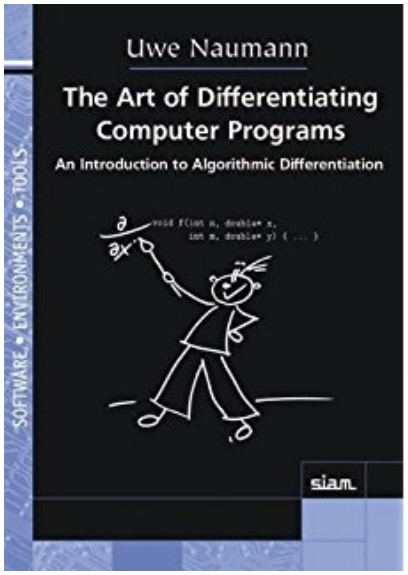


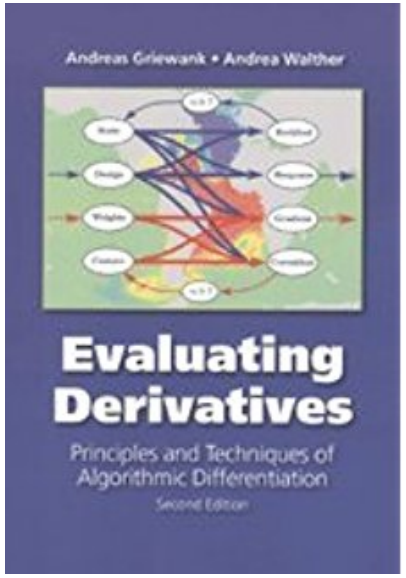
- Actually the name is a misnomer, *this is not a "descent" method*
- But we will stick to it anyway to avoid confusion.
- Performance and parallelism requires batch training
- Larger batch sizes hurt convergence as they get trapped easily
- SGD escapes sharp local minima due to its "noisy" gradients

Training Neural Networks

- Training is performed using an optimization algorithm like SGD
- **SGD needs derivatives.**
- The algorithm ***to compute derivatives*** on a neural network is called ***back-propagation***.
- The back-propagation algorithm is ***not a training algorithm***
- **Idea:** Repeated application of the chain rule from calculus

Back-propagation is just a special case of the ***reverse mode automatic/algorithmic differentiation***





Parallelization Opportunities

1. **Data parallelism**

   Distribute* the input (sets of images, text, audio, etc.)

   a) **Batch parallelism**

   - Distribute each full sample to a different processor
   - *When people mention data parallelism in literature, this is what they mean 99% of the time*

   b) **Domain parallelism**

   - Subdivide samples and distribute parts to processors.

2. **Model parallelism:**

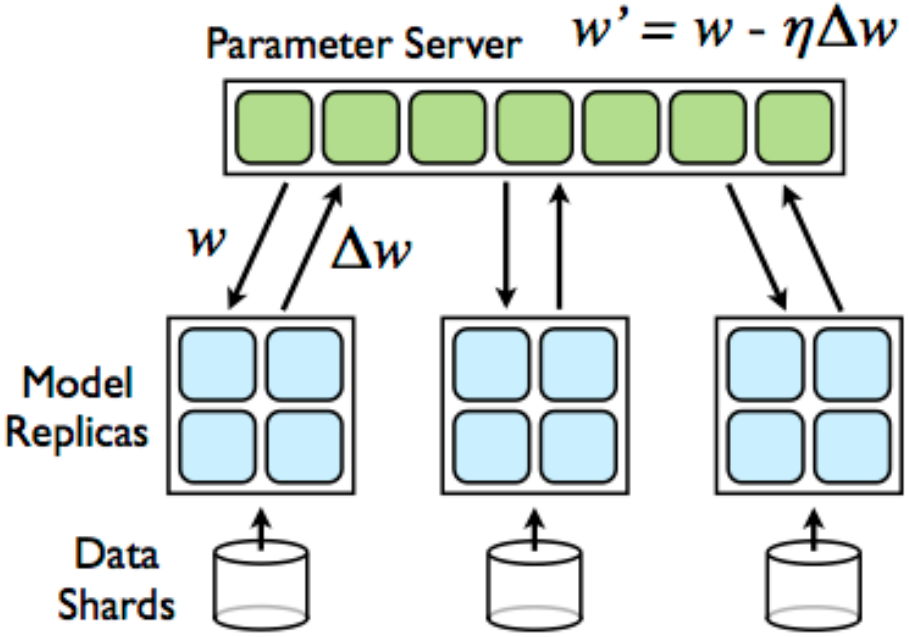
   Distribute the neural network (NN), i.e. its weights

3. **Pipeline parallelism:**

   Inter-batch parallelism, pipelined through NN layers

*: “Distribute” = giving parts to processors (in contrast to replicating)

Batch Parallelism #1



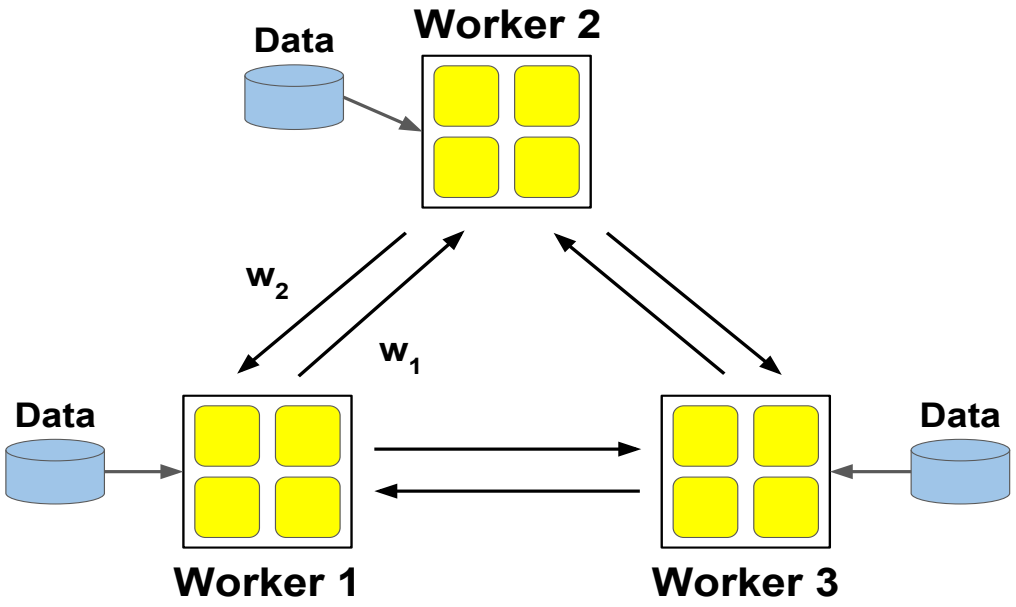
- The fetching and updating of gradients in the parameter server can be done either ***synchronously*** or ***asynchronously***.
- Both has pros and cons. Over-synchronization hurts performance where asynchrony is not-reproducible and might hurt convergence

Dean, Jeffrey, et al. "Large scale distributed deep networks." Advances in neural information processing systems. 2012.

Batch Parallelism #2

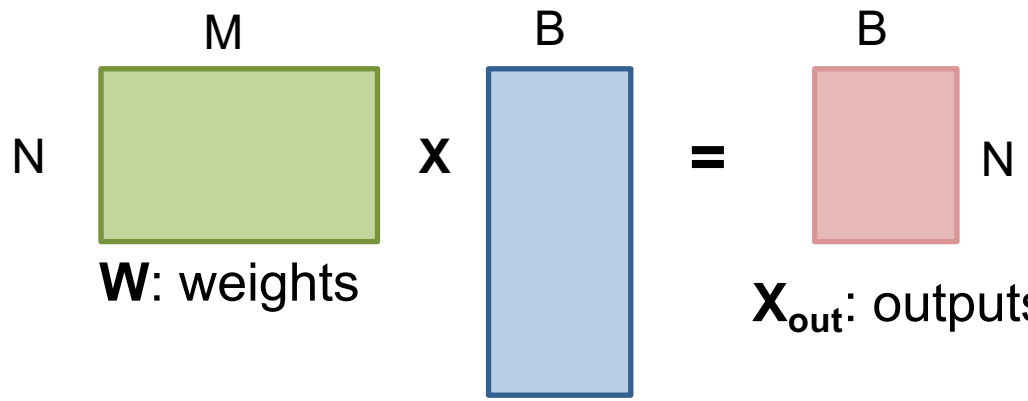
Options to avoid the parameter server bottleneck

1. **For synchronous SGD:** Perform all-reduce over the network to update gradients (good old MPI_Allreduce)
2. **For asynchronous SGD:** Peer-to-peer gossiping



Peter Jin, Forrest Iandola, Kurt Keutzer, "How to scale distributed deep learning?" NIPS ML Sys 2016

Batch Parallel SGD training of NNs as matrix operations

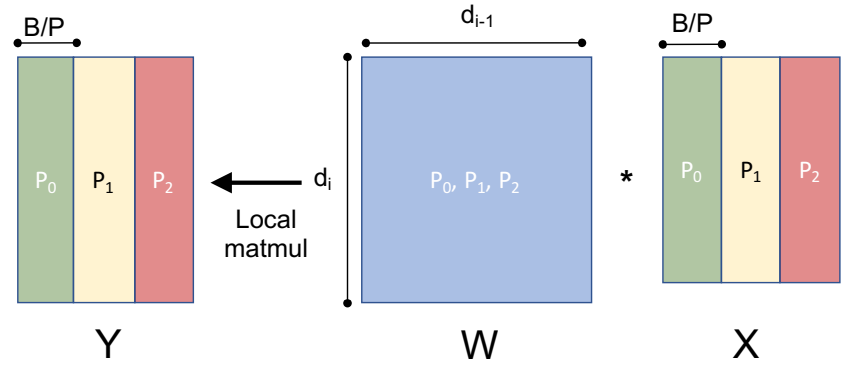


N = the number of outputs
M = the number of inputs
B = the size of the minibatch

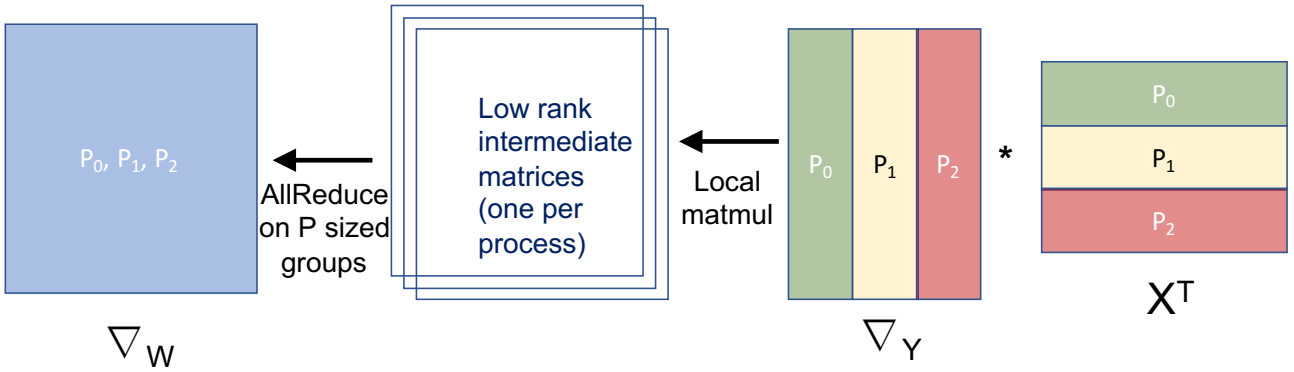
The impact to parallelism:

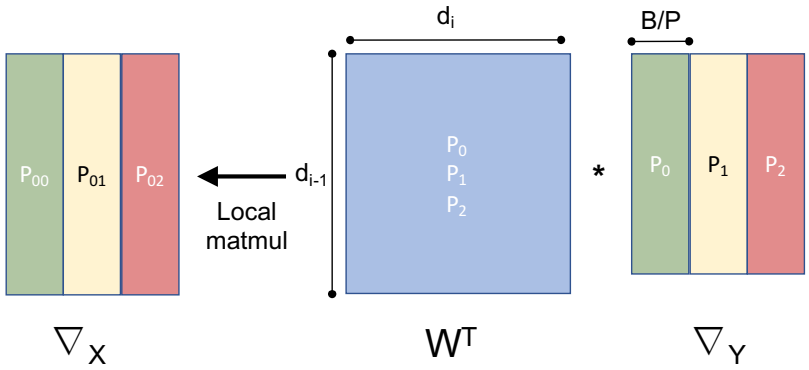
- **W** is replicated to processor, so it doesn't change
- $\mathbf{X_{in}}$ and $\mathbf{X_{out}}$ gets skinnier if we only use data parallelism, i.e. distributing **b=B/p** mini-batches per processor
- GEMM performance suffers as *matrix dimensions get smaller* and *more skewed*
- **Result:** Batch parallelism can hurt single-node performance

Batch Parallel SGD training of NNs as matrix operations



1. Weight matrices (W) are **replicated on each process**
2. Communication only happens during gradient reductions





$\nabla_Y = \partial L / \partial Y$ = how did the loss function change as output activations change?

$\nabla_X = \partial L / \partial X$

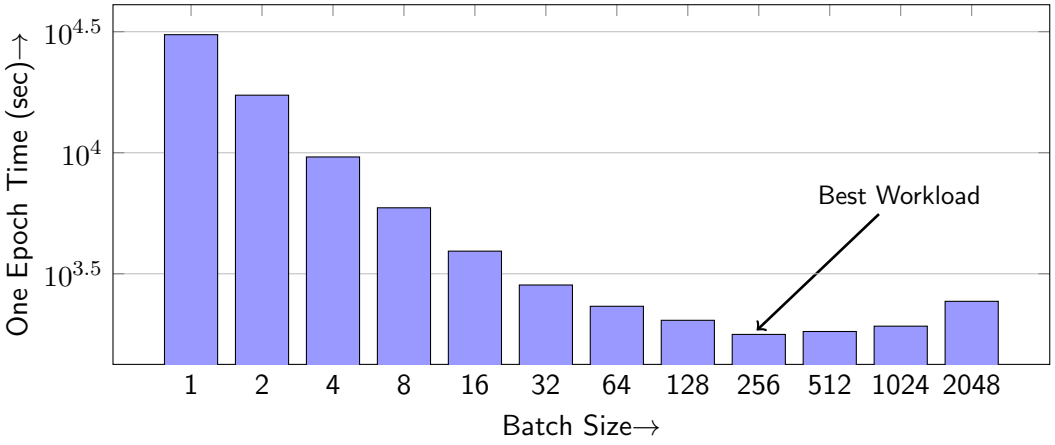
$\nabla_W = \partial L / \partial W$

Batch Parallel Strong Scaling

- **Per-iteration communication cost of batch parallelism is independent of the batch size:**
  - **larger batch → less communication per epoch (full pass over the data set)**

$$T_{comm}(batch) = 2\sum_{i=0}^{L}\left(\alpha\log(P) + \beta\frac{P-1}{P}|W_i|\right)$$

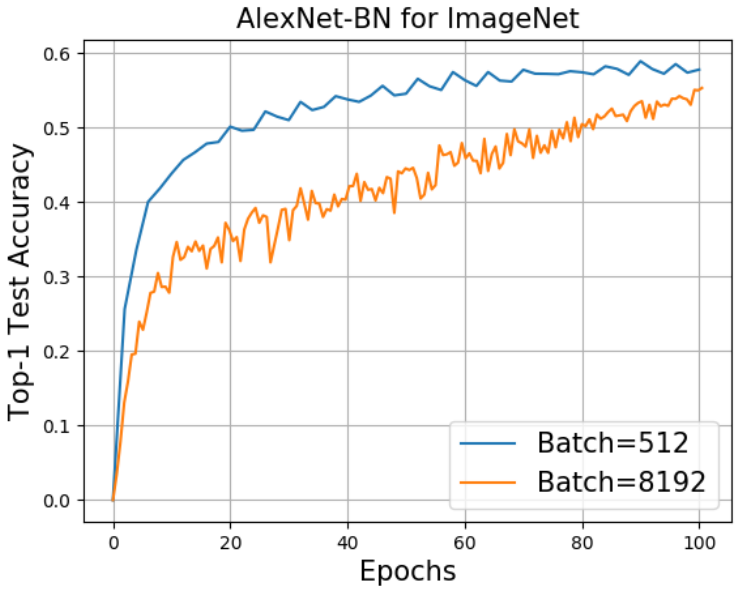
- **But processor utilization goes down significantly for P>>1**
  - **Result: Batch parallel has poor strong scaling**



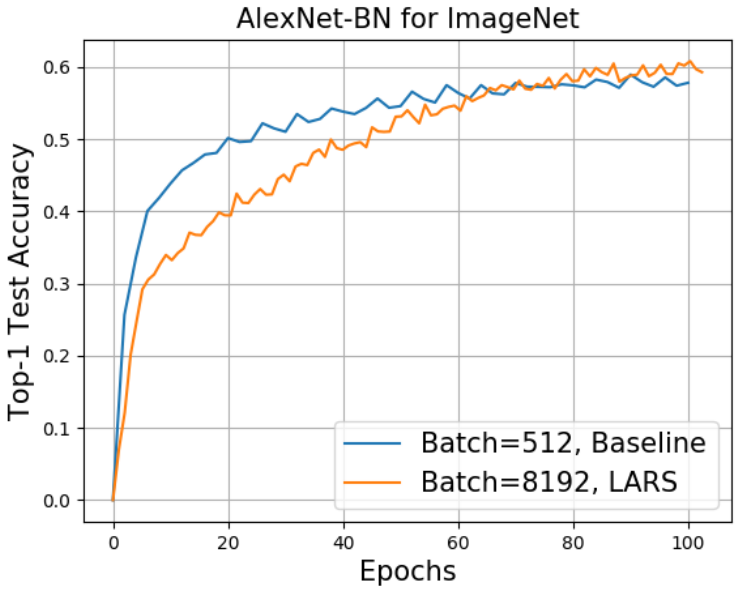
One epoch training time of AlexNet computed on a single KNL

Problems with Batch Parallelism

- **Can not train large models** due to single node memory limitations
- Batch parallel **scaling is limited to batch size B**
  - Larger Batch -> higher strong scaling efficiency
- But vanilla **SGD loses efficiency for large batch sizes**
- Use LARS (Layer-wise Adaptive Rate Scaling) instead



(a) Training without LARS



(b) Training with LARS

Ginsburg, Boris, Igor Gitman, and Yang You. "Large Batch Training of Convolutional Networks with Layer-wise Adaptive Rate Scaling." (2018).

Model Parallelism

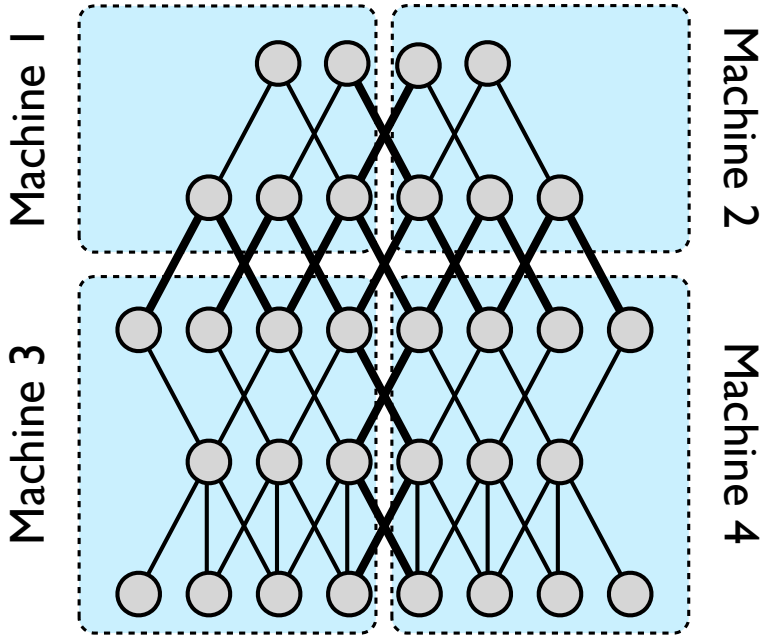
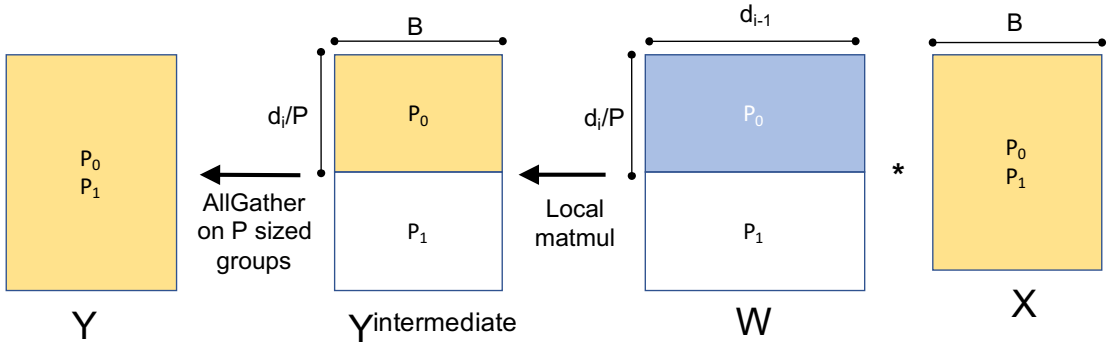


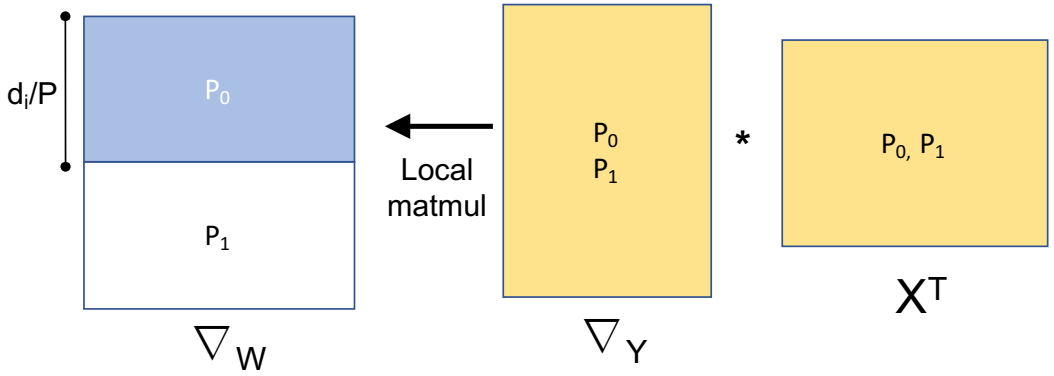
Figure shows both *inter-layer model parallelism* (a.k.a. *pipeline parallelism*) and *intra-layer model parallelism*

Interpretation #1: Partition your neural network into processors
Interpretation #2: Perform your matrix operations in parallel

Dean, Jeffrey, et al. "Large scale distributed deep networks." Advances in neural information processing systems. 2012.

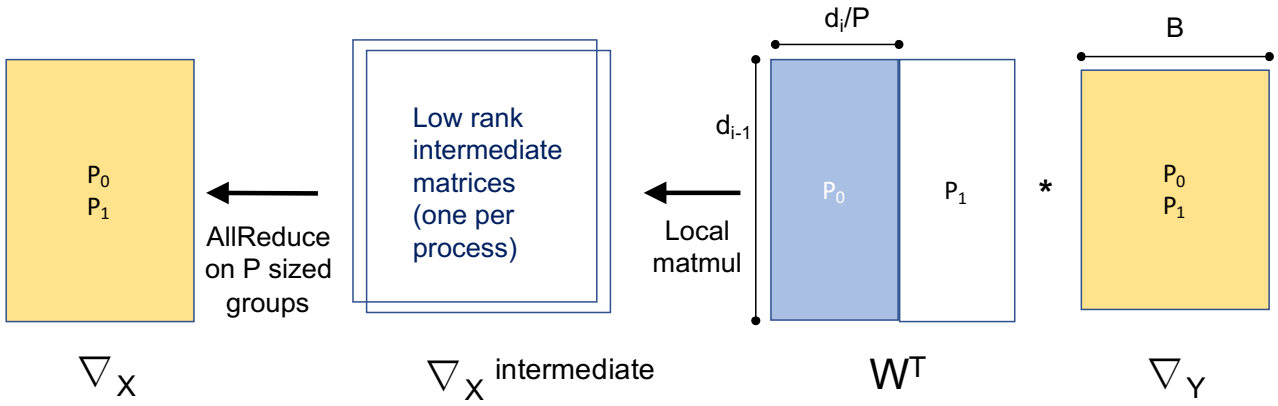
Model Parallel SGD training of NNs as matrix operations





intra-layer model parallelism:

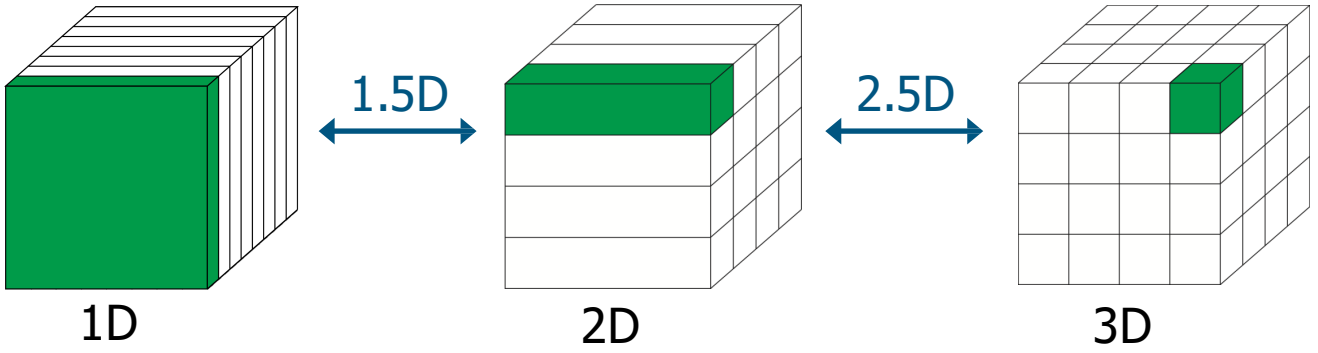
1. Same data passes through all processes, but this is limited to the mini-batch size
2. Two communication steps



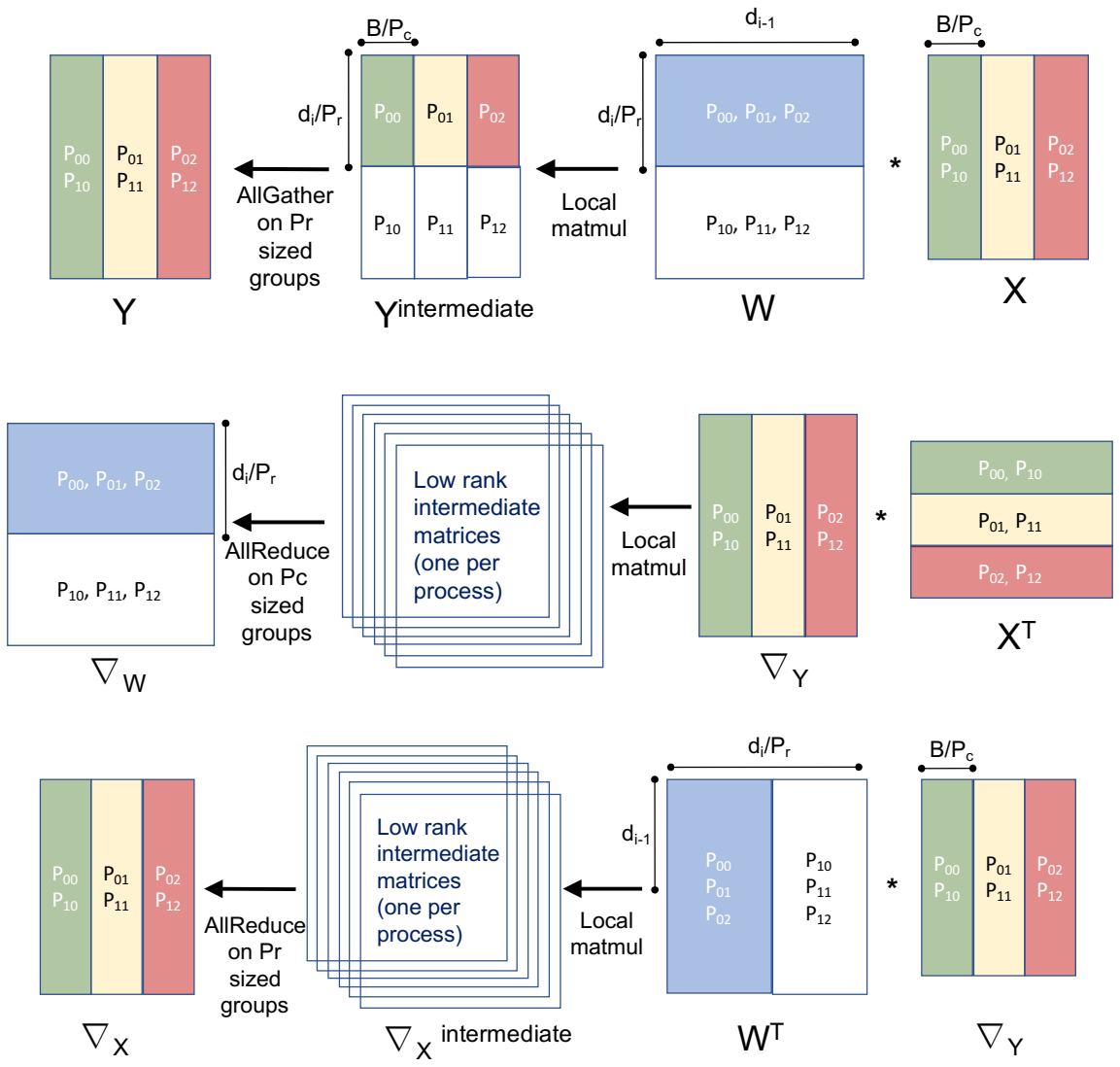
Combinations of various parallelism opportunities

- There are several different ways to combine DNN training parallelism opportunities.
  - It helps to think in terms of matrices again.
- We will exploit ***communication-avoiding matrix algorithms***; which trade off some storage (judicious replication) at the expense of reduced communication.
  - Deep Learning community is already OK with data or model replication in many cases

A succinct classification of parallel matrix multiplication algorithms



Batch & Model Parallel SGD training of NNs as matrix operations

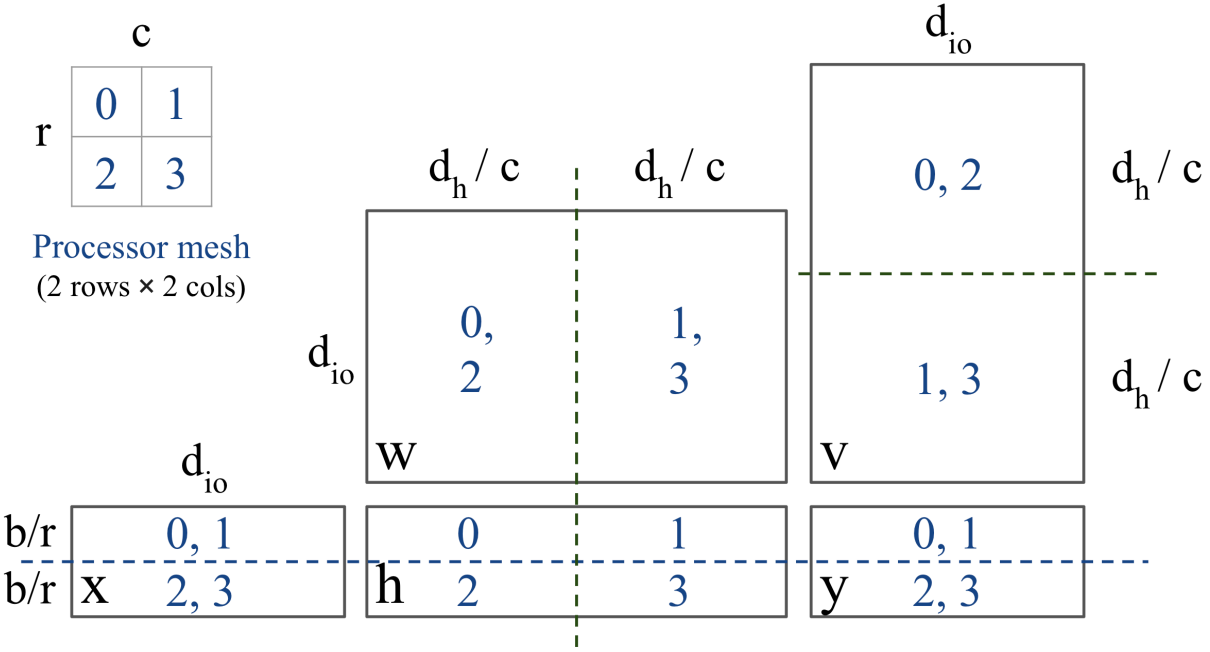


Processes are 2D indexed:
$P = P_r \times P_c$

Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, Aydın Buluç. "Integrated Model, Batch, and Domain Parallelism in Training Neural Networks." ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'18)

How to implement hybrid parallelism

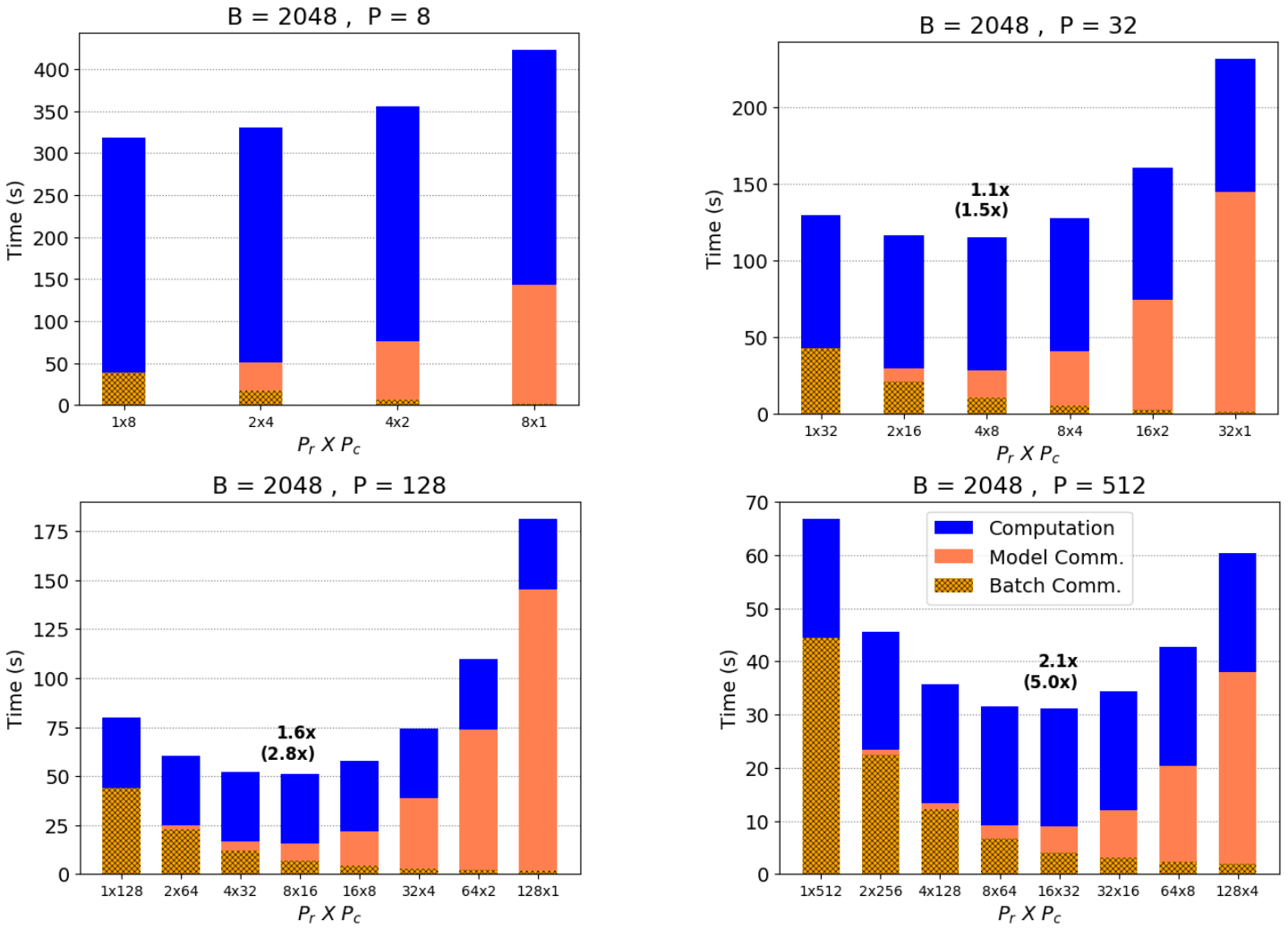
1. Do it yourself manually on PyTorch matrices, if you understand how your preferred hybrid parallelism maps to matrices
2. There are also tools to simplify, such as Mesh-Tensorflow



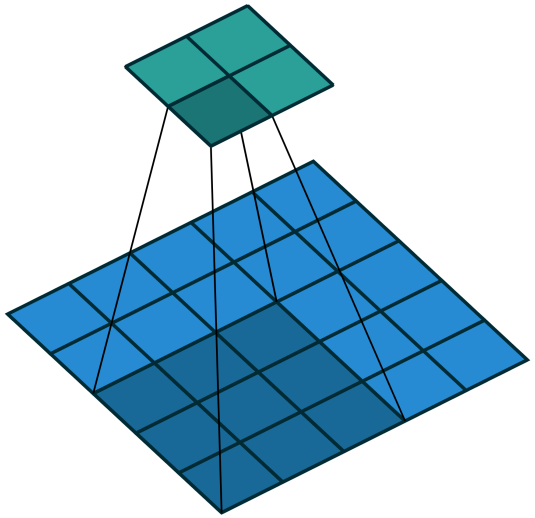
Koanantakool P, et al. Communication-avoiding parallel sparse-dense matrix-matrix multiplication. IPDPS, 2016
Shazeer N, et al. Mesh-TensorFlow: Deep Learning for Supercomputers. NeurIPS. 2018

Integrated Batch + Model Scaling

- **For large processes integrated could provide up to 2x speedup**



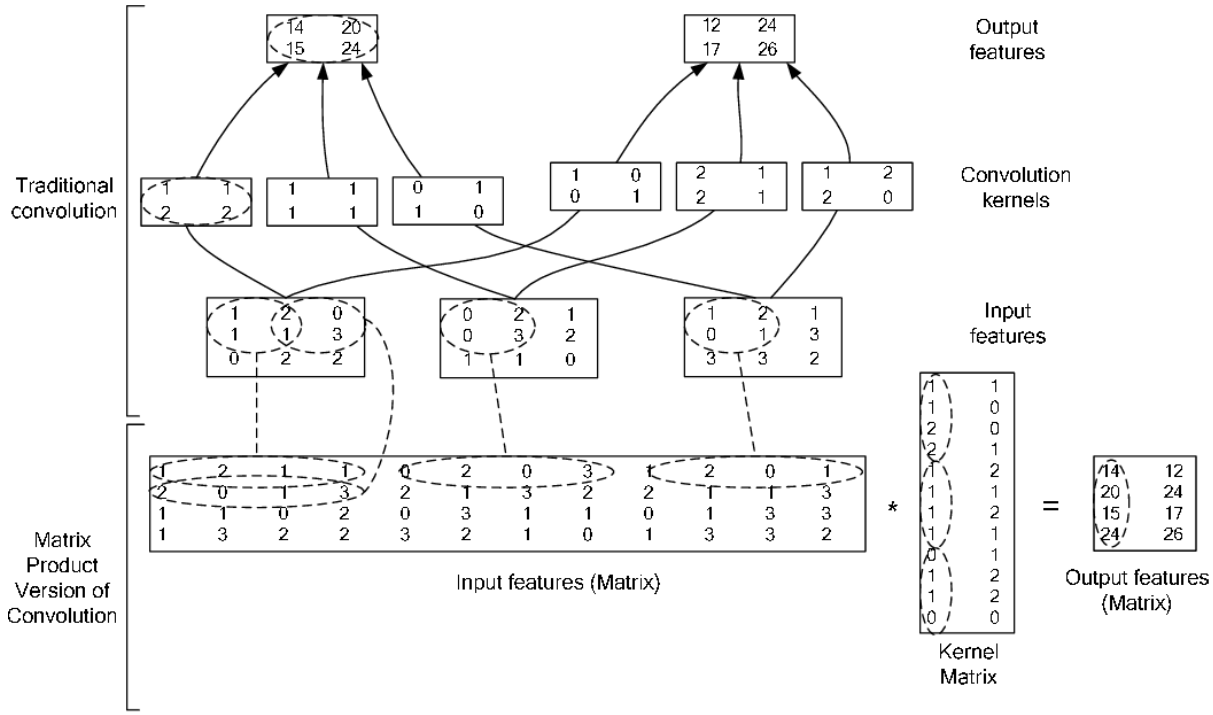
What about convolutional neural networks (CNNs)?



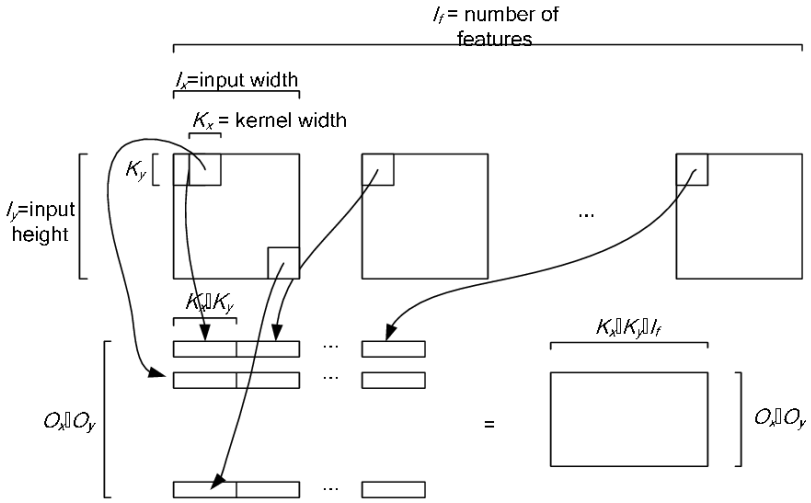
Left: Vincent Dumoulin, Francesco Visin - [A guide to convolution arithmetic for deep learning]

Bottom: Chellapilla, K., Puri, S., & Simard, P. (2006). High performance convolutional neural networks for document processing.

You can convert direct convolutions to matrix multiplies with overhead proportional to the ratio of overlap (function of stride length) to the filter size

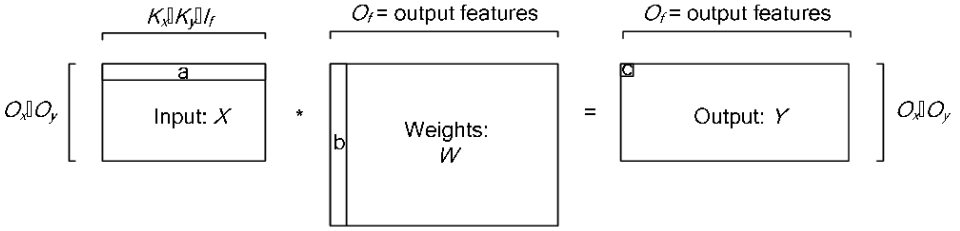


What about convolutional neural networks (CNNs)?

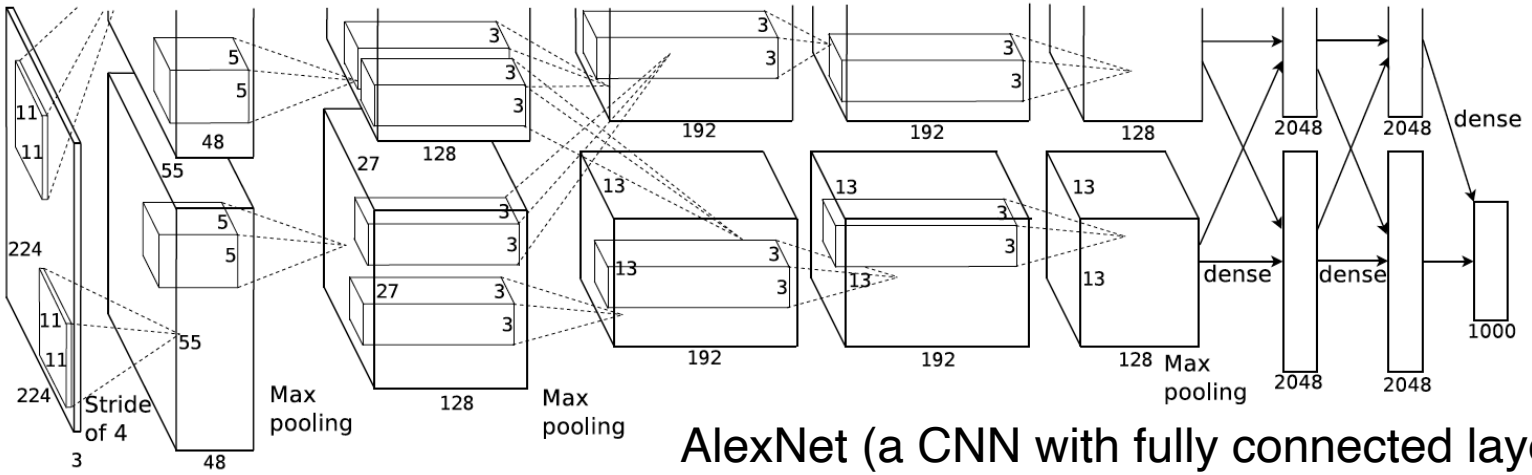


$O_x = ((I_x - K_x + 1)+(S_x - 1))/S_x$ O_x: output width, S_x: horizontal subsampling

$O_y = ((I_y - K_y + 1)+(S_y - 1))/S_y$ O_y: output height, S_y: vertical subsampling



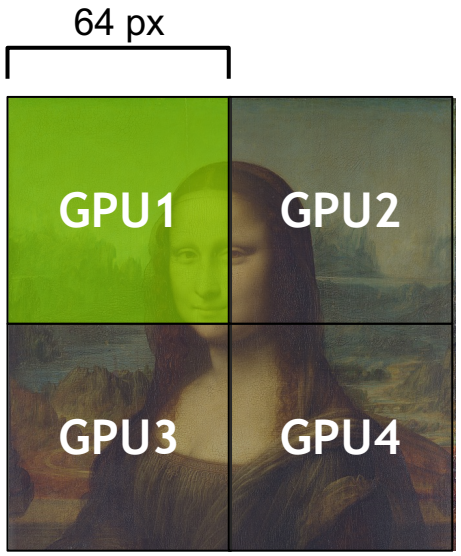
Chellapilla, K., Puri, S., & Simard, P. (2006). High performance convolutional neural networks for document processing.

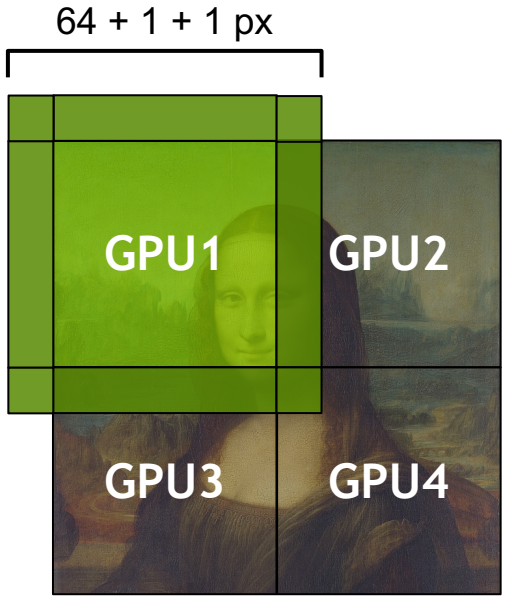


AlexNet (a CNN with fully connected layers)

Domain Parallel

- **The general idea is the same as *halo regions* or *ghost zones* used to parallelize stencil codes in HPC**
  - **Before a convolution, exchange local receptive field boundary data**



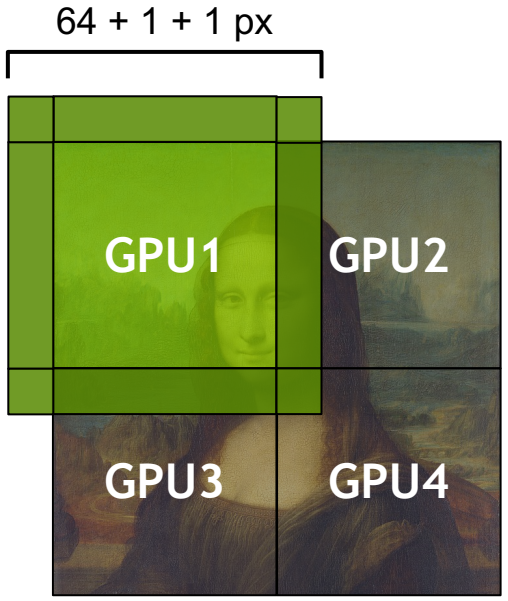


Peter Jin, Boris Ginsburg, and Kurt Keutzer. "Spatially Parallel Convolutions" ICLR Workshop Track, 2018

Communication Complexity of Domain Parallel

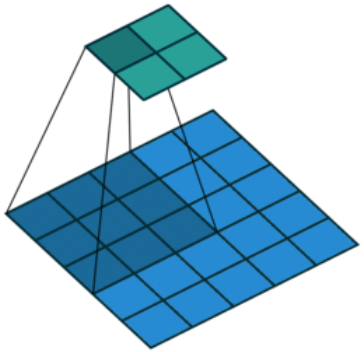
- **Additional communication for halo exchange during forward and backwards pass**
  - **Negligible cost for early layers for which activation size is large (i.e. convolutional)**

$$T_{comm}(domain) = \sum_{i=0}^{L} \left(\alpha + \beta B X_W^i X_C^i k_h^i / 2\right)$$
$$+ \sum_{i=0}^{L} \left(\alpha + \beta B Y_W^i Y_C^i k_w^i / 2\right)$$
$$+ 2 \sum_{i=0}^{L} \left(\alpha \log(P) + \beta \frac{P-1}{P} |W_i|\right)$$



Domain Parallel Scaling

- **Domain parallel scaling on V100 GPUs**
  - **B=32, C=64, K=3, D=1, R=1**



Resolution	GPUs	Fwd. wall-clock	Bwd. wall-clock
128 × 128	1	2.56 ms (1.0×)	6.63 ms (1.0×)
	2	1.52 ms (1.7×)	3.50 ms (1.9×)
	4	**1.23 ms (2.1×)**	**2.33 ms (2.8×)**
256 × 256	1	10.02 ms (1.0×)	26.81 ms (1.0×)
	2	5.34 ms (1.9×)	11.79 ms (2.3×)
	4	**3.11 ms (3.2×)**	**6.96 ms (3.9×)**
512 × 512	1	45.15 ms (1.0×)	126.11 ms (1.0×)
	2	20.18 ms (2.2×)	60.15 ms (2.1×)
	4	**10.65 ms (4.2×)**	**26.76 ms (4.7×)**

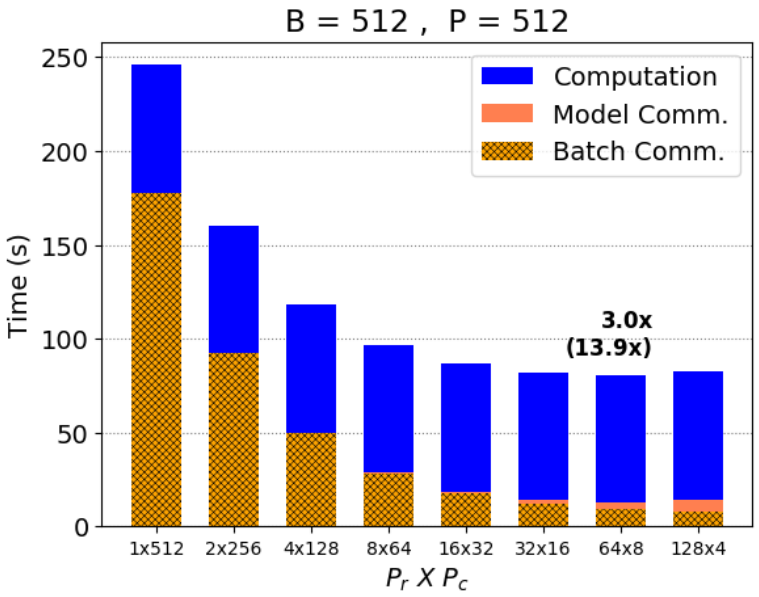
Peter Jin, Boris Ginsburg, and Kurt Keutzer. "Spatially Parallel Convolutions" ICLR Workshop Track, 2018
Figure: Dumoulin, V., Visin, F.. A guide to convolution arithmetic for deep learning. *arXiv:1603.07285*, 2016.

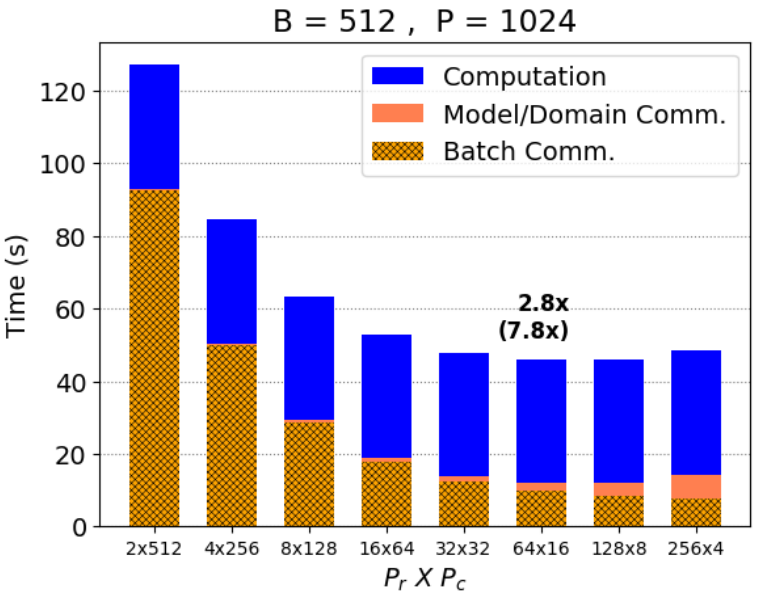
Integrated Batch, Domain, and Model Parallel

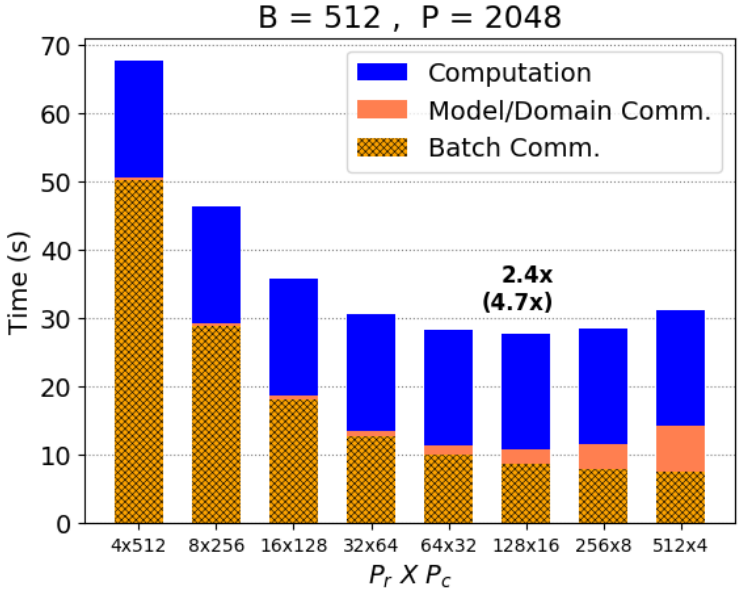
- **Batch + Model for layers with small activation size and large parameters (fully-connected layers and transformer networks)**
- **Batch + Domain for layers with large activation sizes (convolutional layers)**

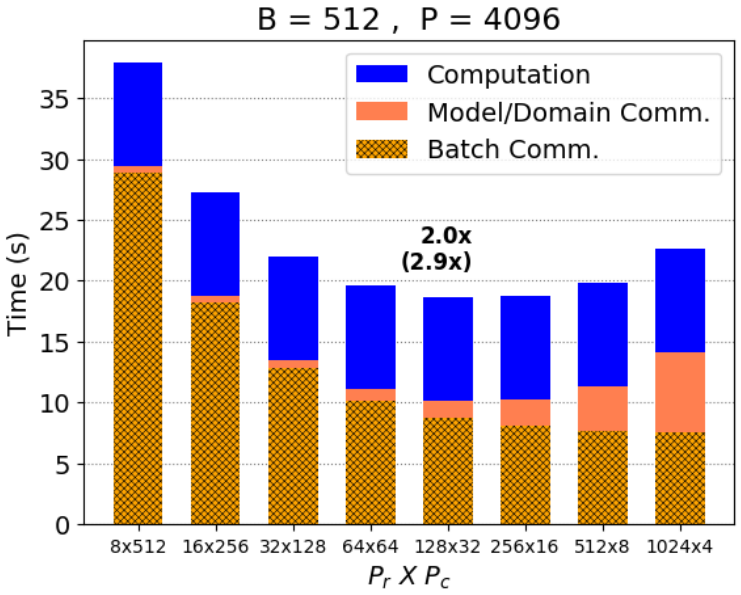
$$
\begin{aligned}
T_{comm} = & \sum_{i \in L_M} \left(\alpha \log(P_r) + \beta \frac{B}{P_c} \frac{P_r - 1}{P_r} d_i \right) + \quad 2 \sum_{i \in L_M} \left(\alpha \log(P_r) + \beta \frac{B}{P_c} \frac{P_r - 1}{P_r} d_{i-1} \right) \\
& + 2 \sum_{i \in L_M} \left(\alpha \log(P_c) + \beta \frac{P_c - 1}{P_c} \frac{|W_i|}{P_r} \right) + \quad \sum_{i \in L_D} \left(\alpha + \beta \frac{B}{P_c} X_W^i X_C^i k_h^i / 2 \right) \\
& + \sum_{i \in L_D} \left(\alpha + \beta \frac{B}{P_c} X_W^{i+1} X_C^{i+1} k_w^i / 2 \right) + \quad 2 \sum_{i \in L_D} \left(\alpha \log(P) + \beta \frac{P - 1}{P} |W_i| \right)
\end{aligned}
$$

Integrated Batch, Domain, and Model Parallel

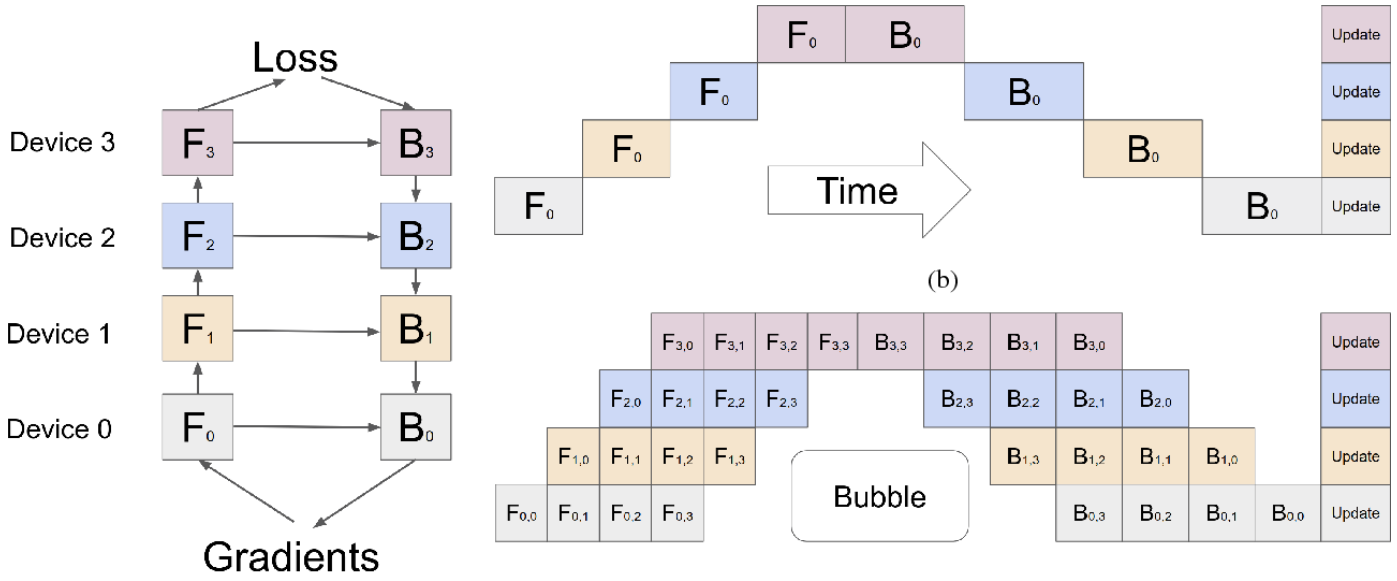








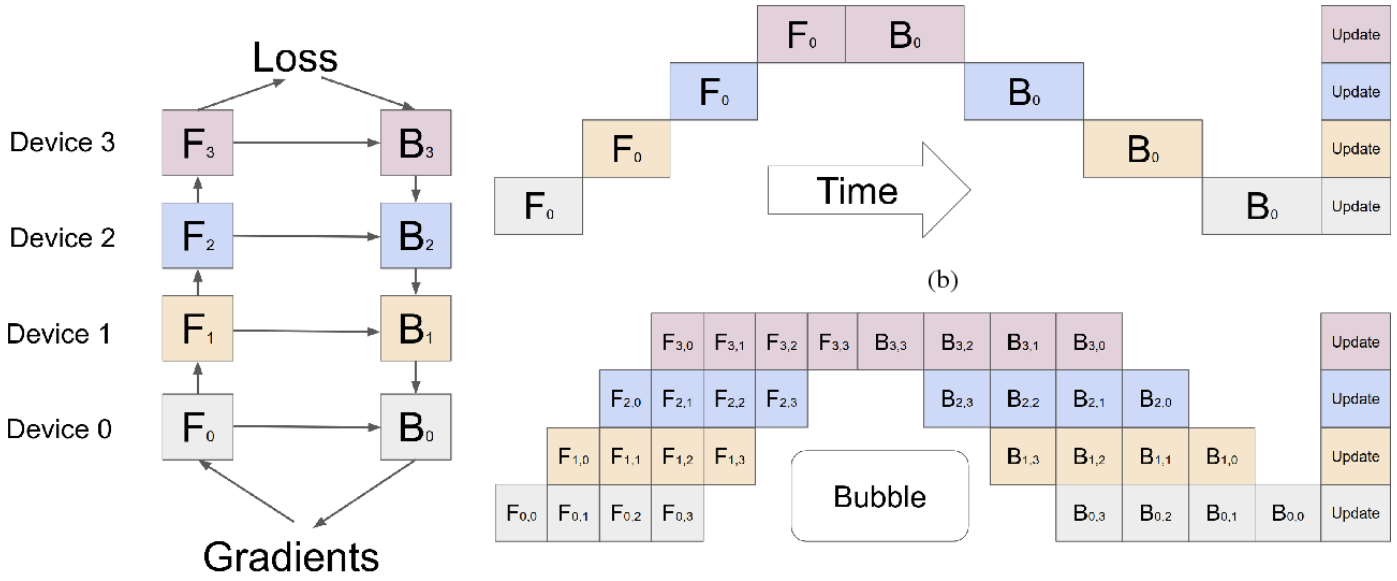
Pipeline Parallelism



- Pipeline parallelism divides the input mini-batch into **smaller micro-batches**, enabling different GPUs to work on different micro-batches simultaneously. Gradients are applied synchronously at the end.
- Without micro-batching, pipelining would not expose any real parallelism (i.e., layers would merely be processed by different GPUs)

- Petrowski A, Dreyfus G, Girault C. Performance analysis of a pipelined backpropagation parallel algorithm. IEEE Transactions on Neural Networks. 1993 Nov;4(6):970-81 (original idea)
- Huang Y, Cheng Y, Chen D, Lee H, Ngiam J, Le QV, Chen Z. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. NeurIPS, 2019 (figure source)

Pipeline Parallelism



- Pipeline parallelism is a mix of (inter-layer) **model parallelism,** as *it parallelizes across the inter-layer NN model structure*, and **batch parallelism,** as *it needs micro-batches of data for filling the pipeline*.
- **Pipeline bubbles:** start of the forward propagation on a minibatch requires the backprop of the previous minibatch to complete
- Contribution of pipeline parallelism to the total available parallelism is **a multiplicative factor that is bounded by the NN depth**. Without a deep network, all micro-batching would achieve is batch parallelism.

Pipeline parallelism: synchronization of the weight matrix

In the backpropagation, you need to compute gradients using the same weight matrix W you used during forward propagation:

1. Either each process can store its own W: W_1, W_2,…, W_p, effectively increasing memory footprint to match batch parallelism
2. Or we do periodic flushes so we can use one synchronized W.

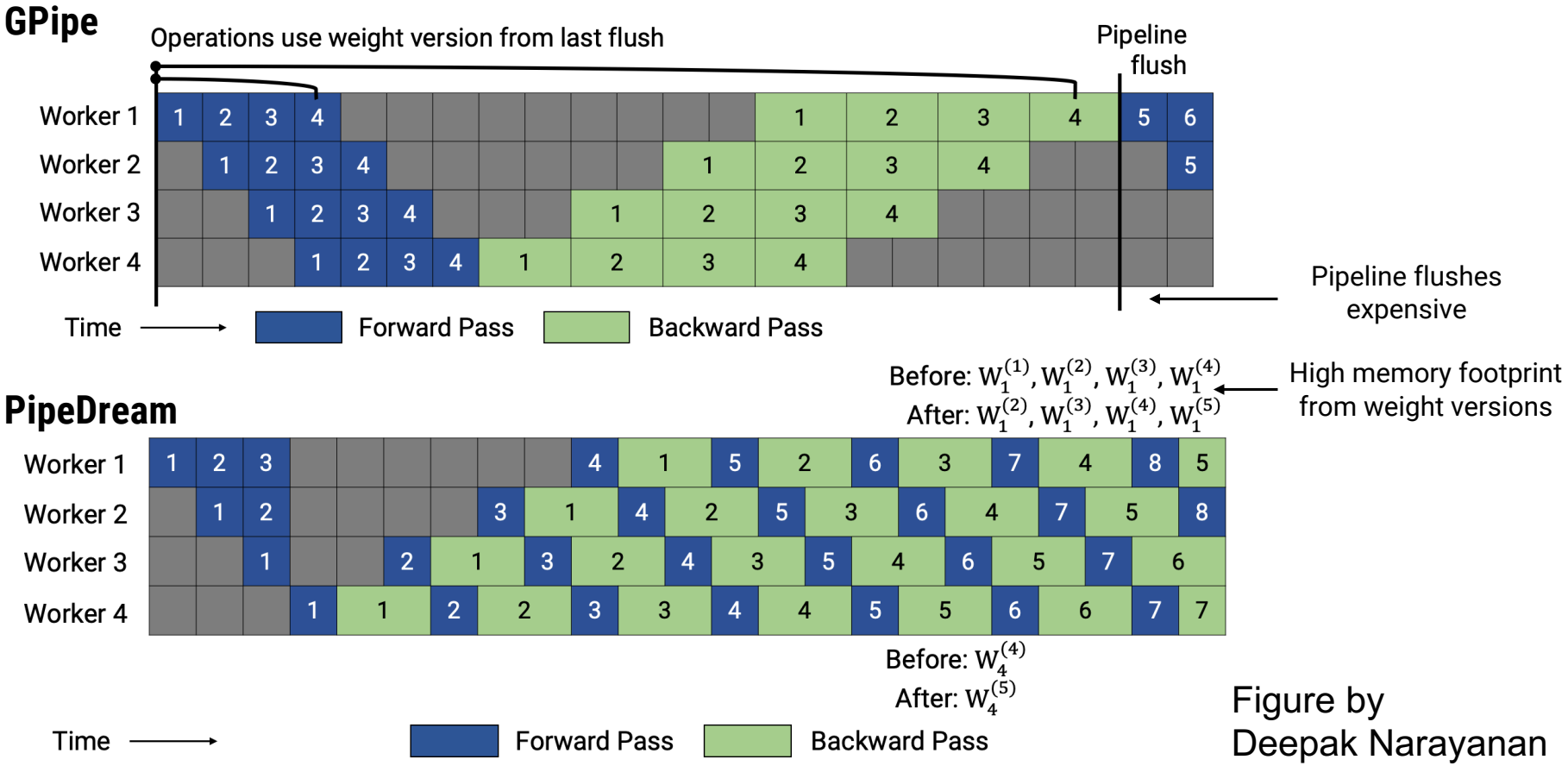


Figure by Deepak Narayanan

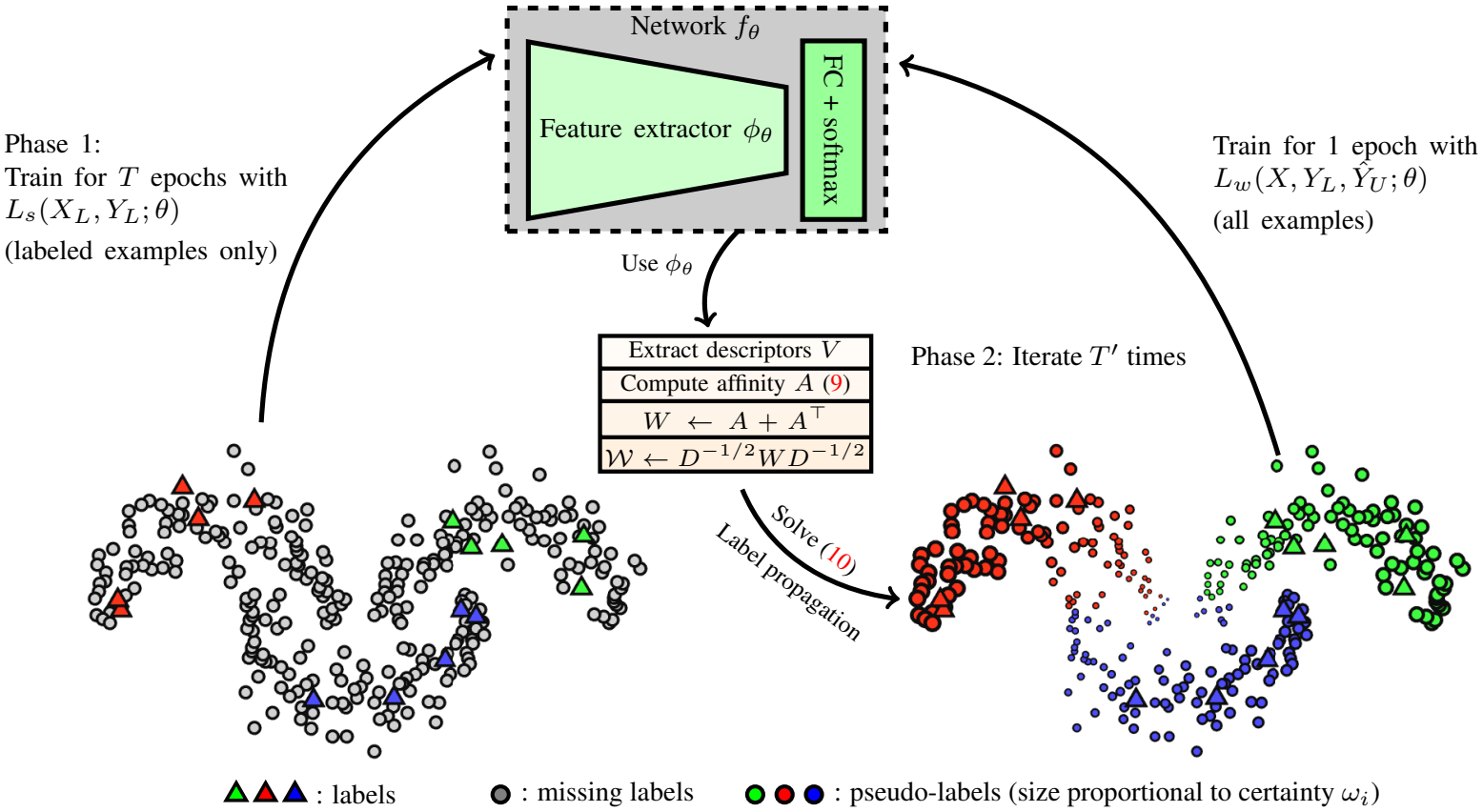
Summary of distributed deep learning

- Large batch size training often lead to suboptimal learning, but this can be mitigated with better learning algorithms such as LARS
- Integrated parallelism uses communication-avoiding algorithms to extend scaling beyond batch size
- Integrated parallelism optimally combines model and data (batch and domain) parallelism and often performs better than each extreme.
- It is often better [1] to use (inter-layer) **model parallelism** for fully connected layers (large parameters), and **batch parallelism** for convolutional layers (large activations)
- Pipeline parallelism can also be combined (in principle) with intra-layer model parallelism.

[1] Krizhevsky, Alex. "One weird trick for parallelizing convolutional neural networks." *arXiv preprint arXiv:1404.5997* (2014).

Semi-supervised Learning

- A (small) subset of data has training labels, but most of the data is unlabeled
- *Label propagation* is the canonical graph semi-supervised learning algorithm



Motivation for Graph Neural Networks

"GNNs are among the most general class of deep learning architectures currently in existence, […] and most other deep learning architectures can be understood as a special case of the GNN with additional geometric structure"

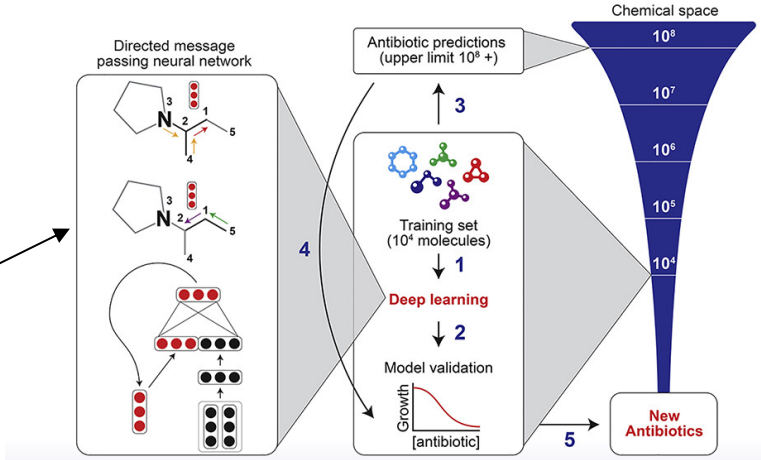
Bronstein, Michael M., et al. "Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges." (2021)

NEWS · 20 FEBRUARY 2020

Powerful antibiotics discovered using AI

Machine learning spots molecules that work even against 'untreatable' strains of bacteria.

This is a graph neural network



Article | Published: 09 June 2021

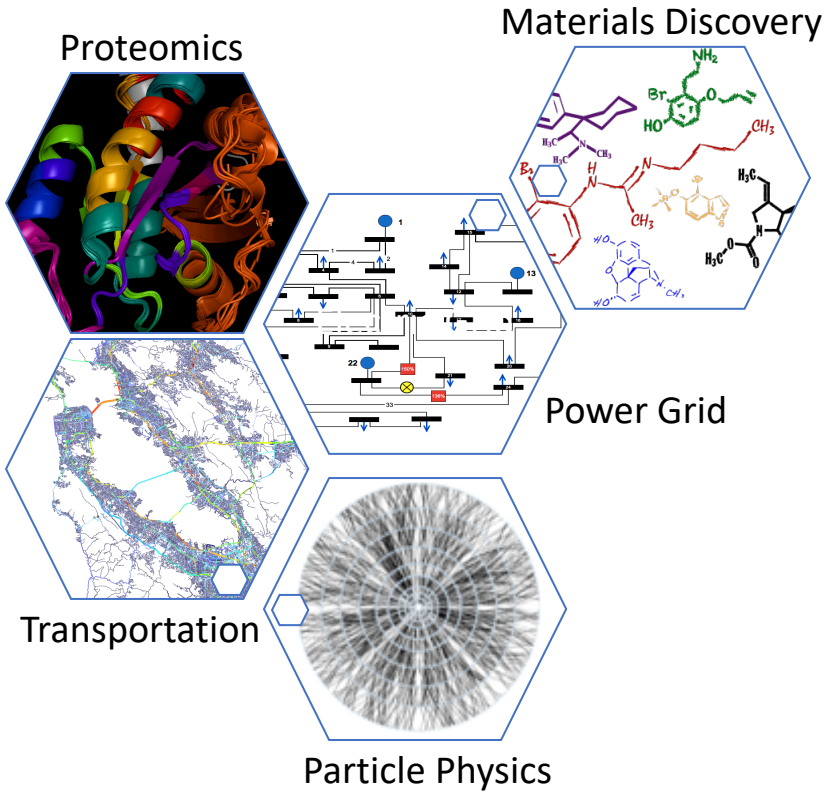
A graph placement methodology for fast chip design

Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, Jiwoo Pak, Andy Tong, Kavya Srinivasa, William Hang, Emre Tuncer, Quoc V. Le, James Laudon, Richard Ho, Roger Carpenter & Jeff Dean

Nature **594**, 207–212 (2021) | Cite this article

… we pose chip floorplanning as a reinforcement learning problem, and develop an **edge-based graph convolutional neural network** architecture…

Graph Neural Networks (GNNs)



GNNs are finding success in many challenging scientific problems that involve interconnected data.

- Graph classification
- Edge classification
- **Node classification**

GNNs are computationally intensive to train. Distributed training need to scale to large GPU/node counts despite challenging sparsity.

How to use GNNs

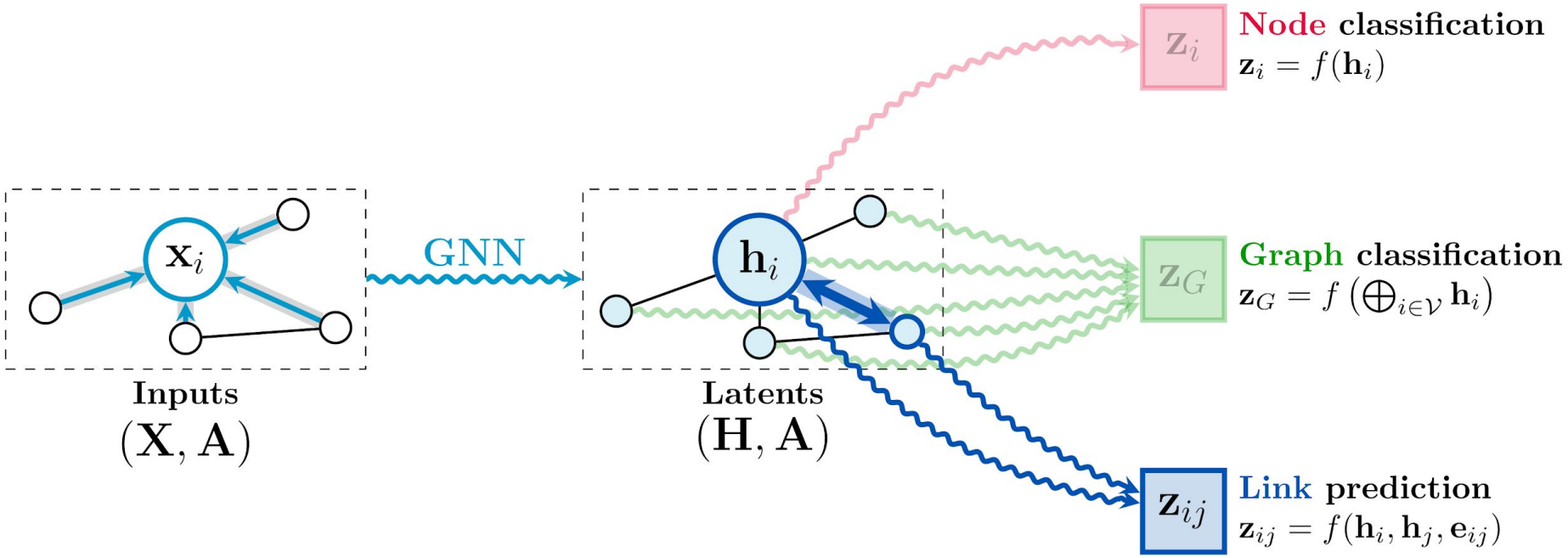
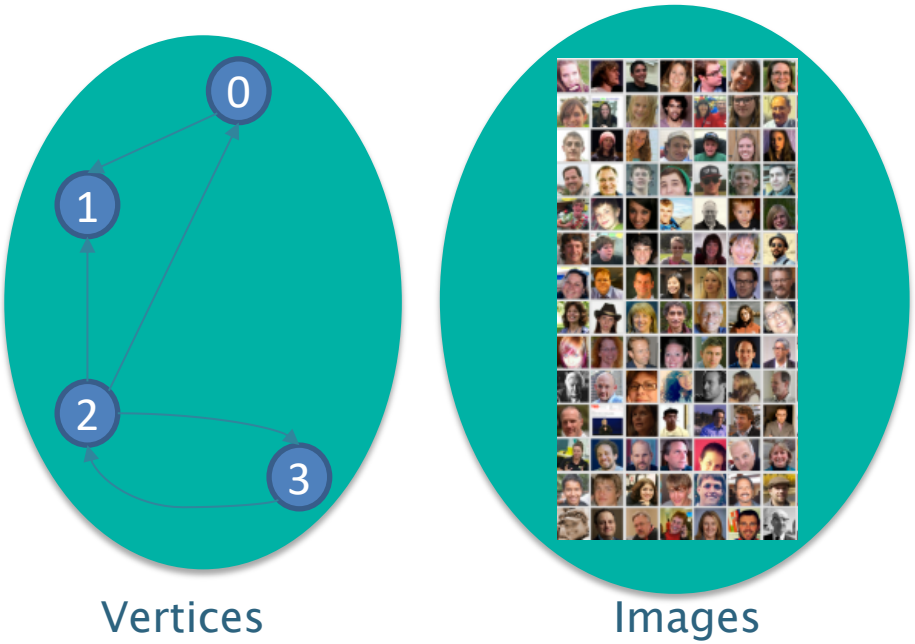


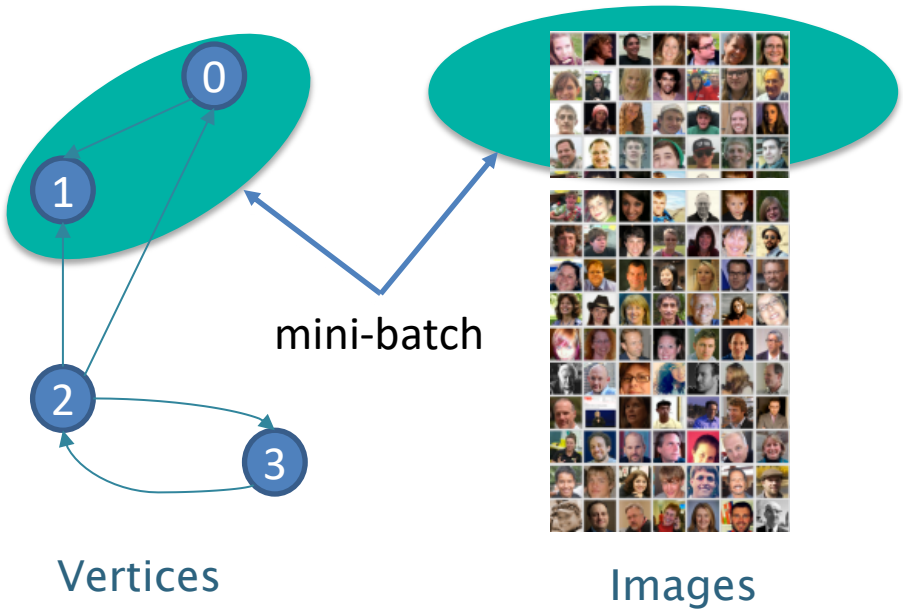
Figure source: Petar Veličković

Full-graph vs. mini-batch SGD



Full-graph training:

- Train on **entire** training set
- Slower convergence per epoch
- Faster training per epoch
- Large memory footprint
- Focus of this presentation



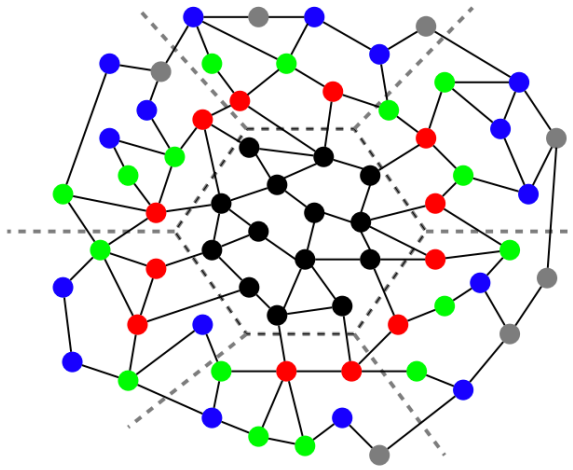
Mini-batch SGD:

- Train on a batch of **samples**
- Faster convergence per epoch
- Slower training per epoch
- Requires graph sampling
- Potentially smaller memory footprint
- Focus of future work

Full-graph vs. mini-batch SGD

No dependencies

sample

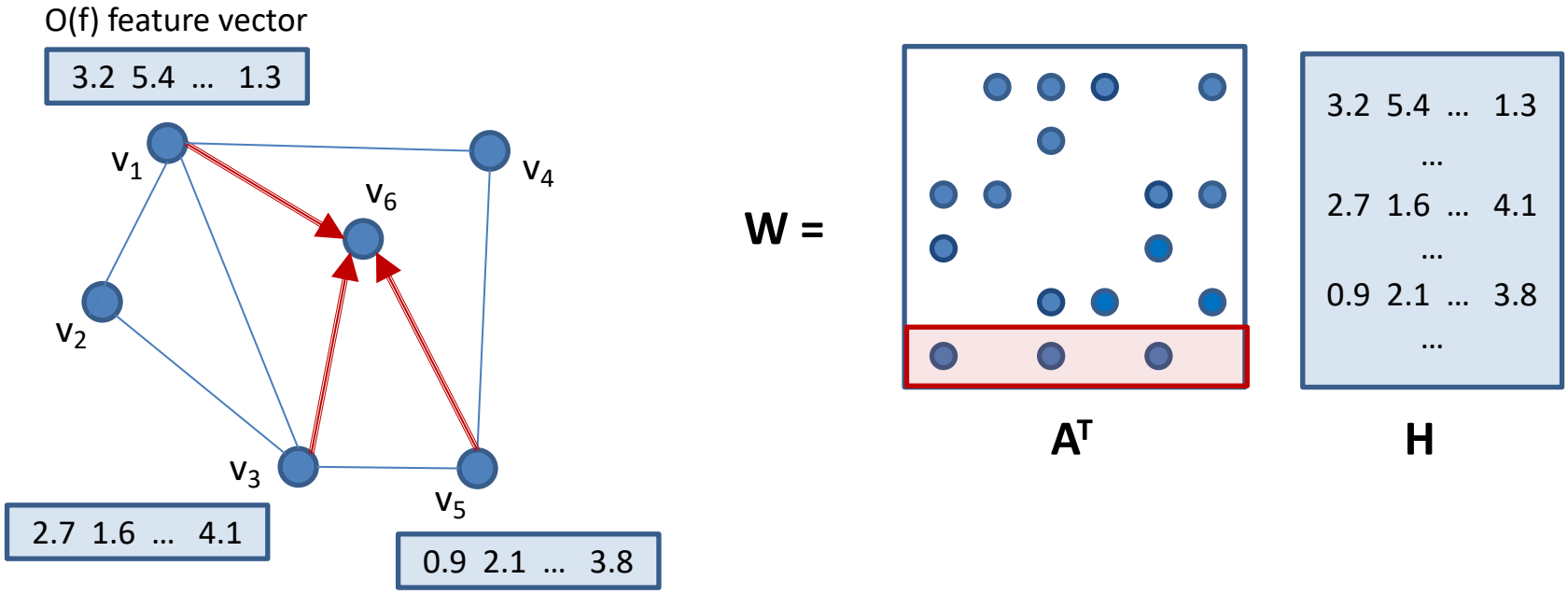


Layered dependencies

- Vertices (unlike images) are dependent on each other
- L-layer GNN uses L-hop neighbors for vertices in batch
- Must store almost the whole graph for any minibatch for power-law graphs
- How to subsample from L-hop neighborhood and keep accuracy?
- CAGNET (Communication-Avoiding Graph Neural nETworks) full gradient descent to avoid such issues: https://github.com/PASSIONLab/CAGNET/

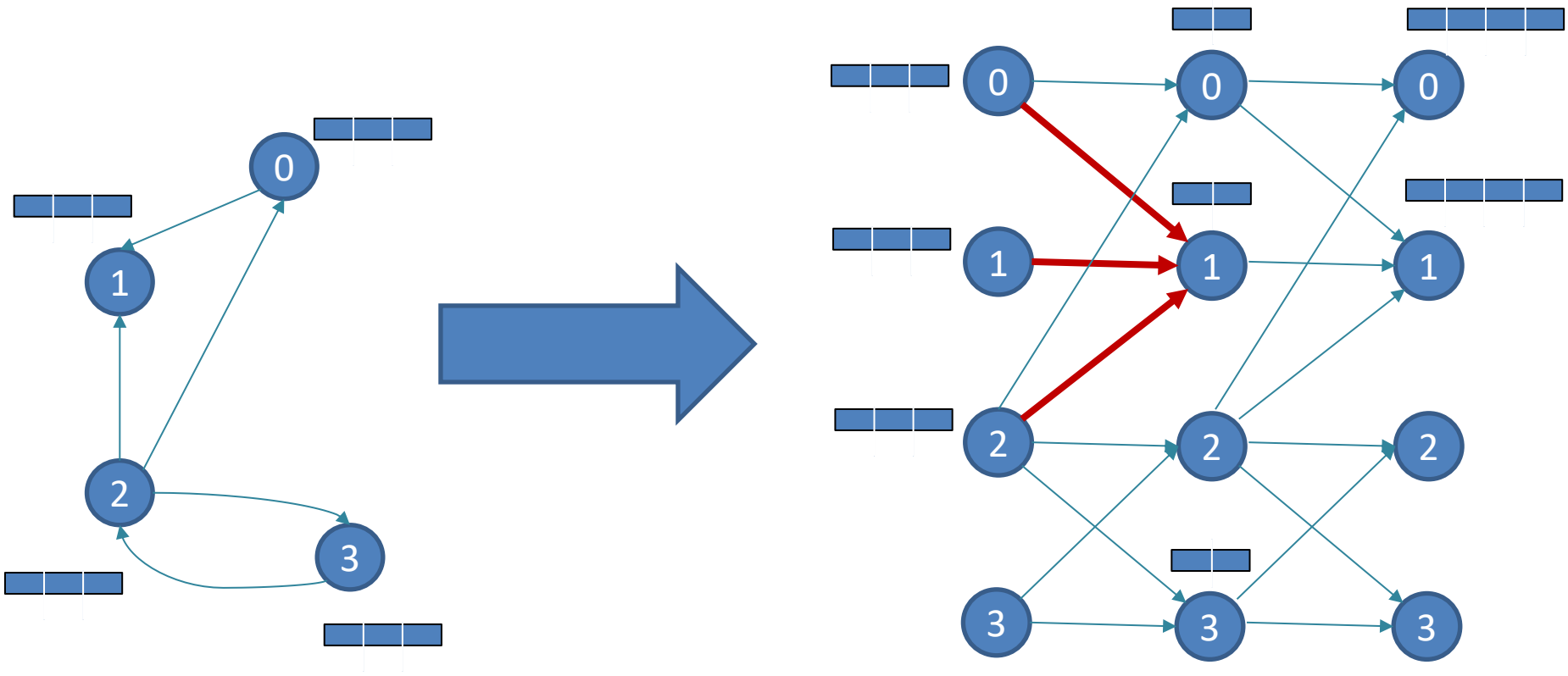
Graph convolutions

Graph convolution: Feature aggregation from neighbors



- GNN is an umbrella term for any neural network that performs graph representation learning.
- CAGNET focuses on Graph Convolutional Networks (GCNs)
- We are working on adding graph attention layers

Graph convolutions



Input Graph

GNN of Input Graph

- Recall that a CNN can have different *channel* dimension at each layer.
- GNNs also have different embedding dimension at each layer

GCN training in matrix notation

Forward Propagation:

$$\mathbf{Z}^l \leftarrow \mathbf{A}^\mathsf{T}\mathbf{H}^{l-1}\mathbf{W}^l$$

$$\mathbf{H}^l \leftarrow \sigma(\mathbf{Z}^l)$$

Backward Propagation:

$$\mathbf{G}^{l-1} \leftarrow \mathbf{A}\mathbf{G}^l(\mathbf{W}^l)^\mathsf{T} \odot \sigma'(\mathbf{Z}^{l-1})$$

$$\mathbf{Y}^{l-1} \leftarrow (\mathbf{H}^{l-1})^\mathsf{T}\mathbf{A}\mathbf{G}^l$$

Symbols and Notations	
Symbol	Description
$\mathbf{A}$	Modified adjacency matrix of graph $(n \times n)$
$\mathbf{H}^l$	Embedding matrix in layer l $(n \times f)$
$\mathbf{W}^l$	Weight matrix in layer l $(f \times f)$
$\mathbf{Y}^l$	Matrix form of $\frac{\partial \mathcal{L}}{\partial W_{ij}^l}$ $(f \times f)$
$\mathbf{Z}^l$	Input matrix to activation function $(n \times f)$
$\mathbf{G}^l$	Matrix form of $\frac{\partial \mathcal{L}}{\partial Z_{ij}^l}$ $(n \times f)$
σ	Activation function
f	Length of feature vector per vertex
f_u	Feature vector for vertex u
L	Total layers in GNN
P	Total number of processes
α	Latency
β	Reciprocal bandwidth

GCN training

- Each node is initialized with a feature vector
  - H^0 has initial feature vector per node $(n\ x\ f)$
- Each node aggregates vectors of its neighbors, applies a weight
- Each layer computes gradients

```
for i = 1 … E
    for l = 1 … L
        Z^l = A^T * H^(l-1) * W^l
        H^l = σ(Z^l)
    ...
    for l = L-1 … 1
        G^l = A * G^(l+1) * (W^(l+1))^T ⊙ σ'(Z^l)
       dH/dW = (H^(l-1))^T * A * G^l
```

$A \in n\ x\ n$

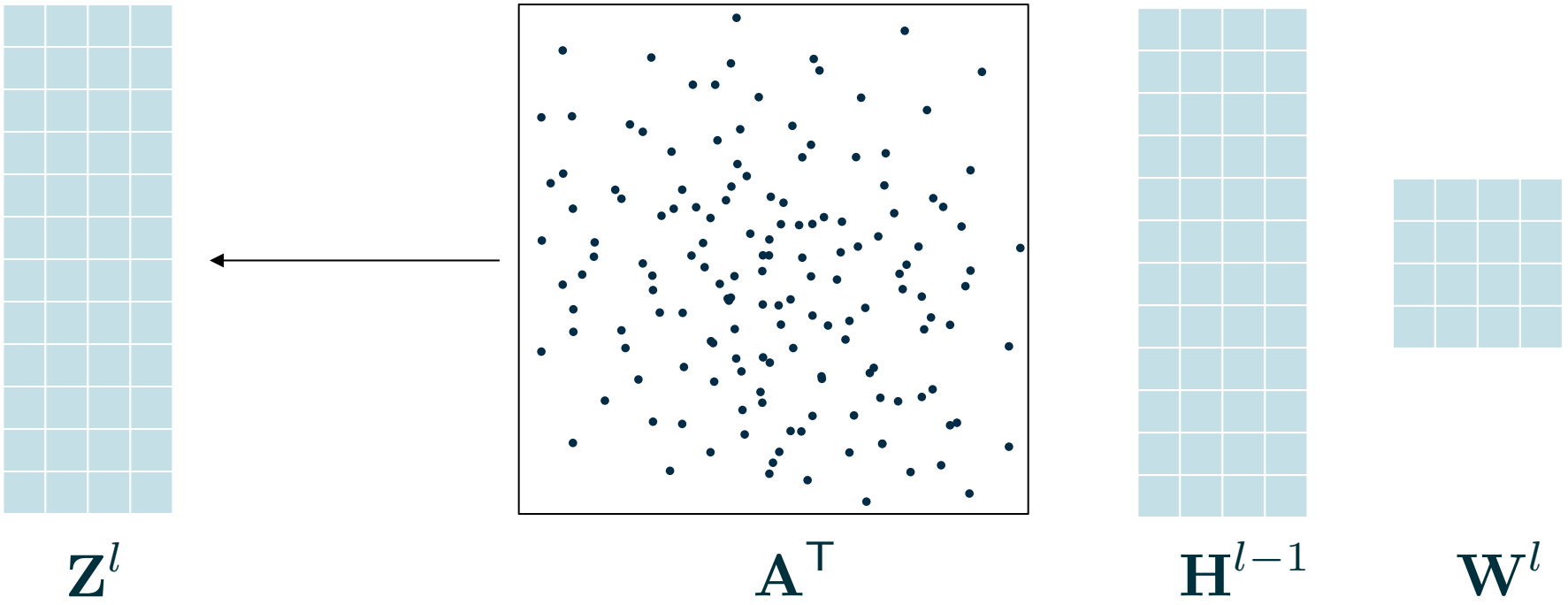
$H^l \in n\ x\ f^l$

$G^l \in n\ x\ f^l$

$W^l \in f^{l-1}\ x\ f^l$

- A is sparse and f << n, so the main workhorse is SpMM (sparse matrix times tall-skinny dense matrix)

Bottleneck of full-graph GCN training

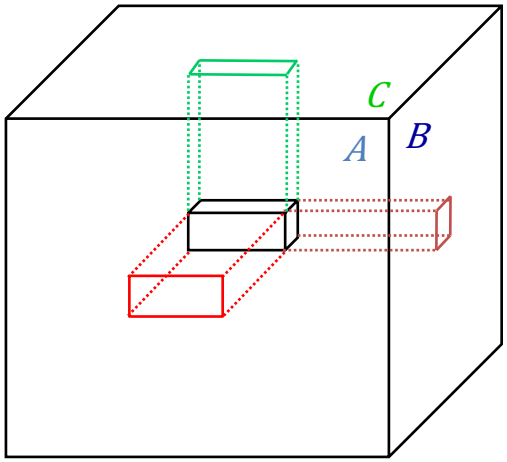


Cost(SpMM) >>> Cost(DGEMM)

(mostly because W is so small)

The computation cube of matrix-matrix multiplication

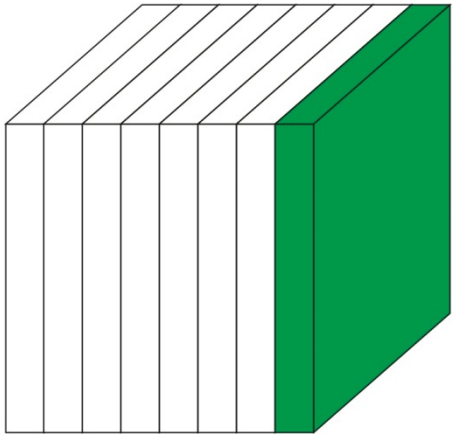
Matrix multiplication: $\forall(i,j) \in n \times n, \quad C(i,j) = \Sigma_k A(i,k)B(k,j),$



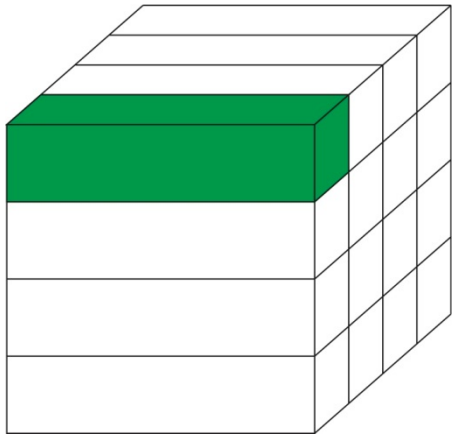
The *computation (discrete) cube*:

- A face for each (input/output) matrix
- A grid point for each multiplication

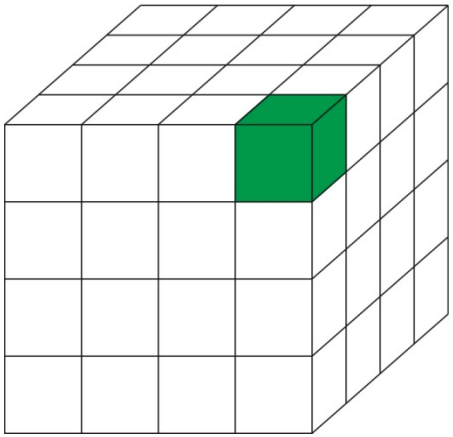
.5D algorithms interpolate between two



1D algorithms

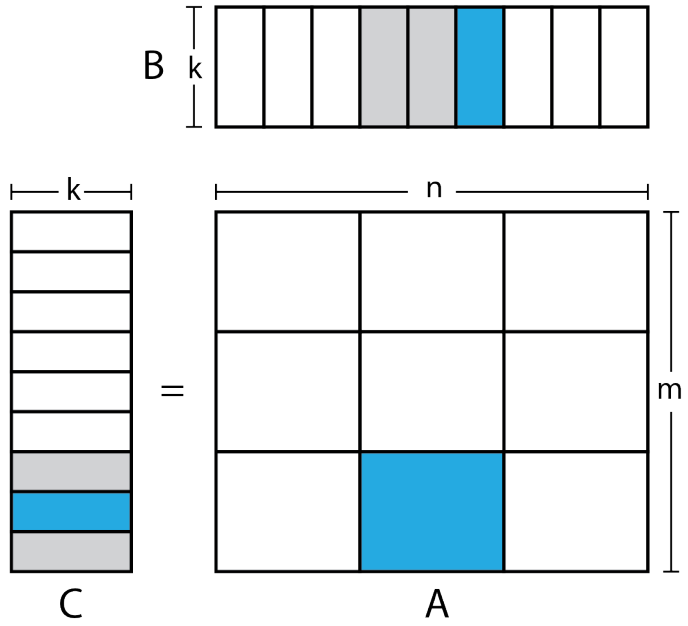


2D algorithms



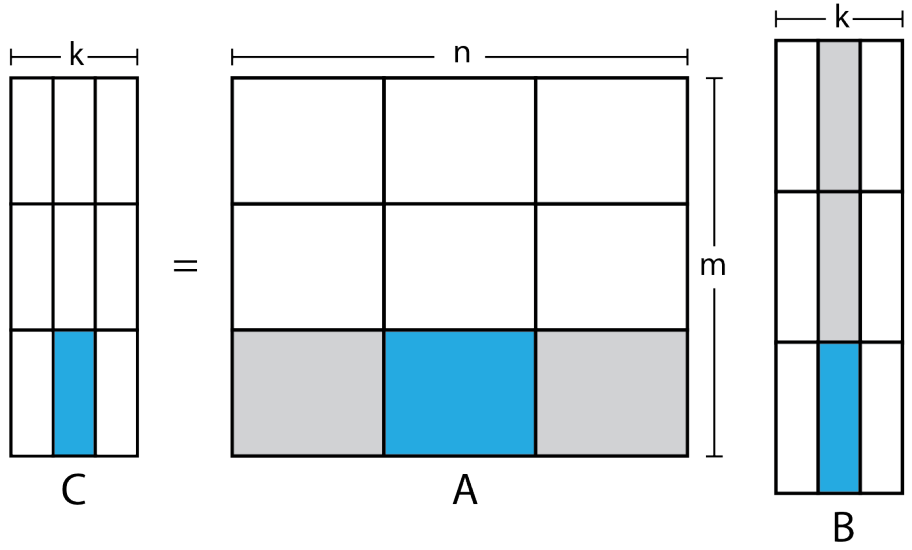
3D algorithms

Distributed SpMM algorithms



- Stationary A, 1.5D algorithm
- **A** is split on a p/c-by-c grid

A is sparse, **B** and **C** are dense



- Stationary C, 2D algorithm
- Memory optimal

- 1D algorithm not shown, degeneration of sA-1.5D for the c=1 case
- Right before reduction, sA-1.5D uses c times more dense-matrix memory

Distributed SpMM algorithms

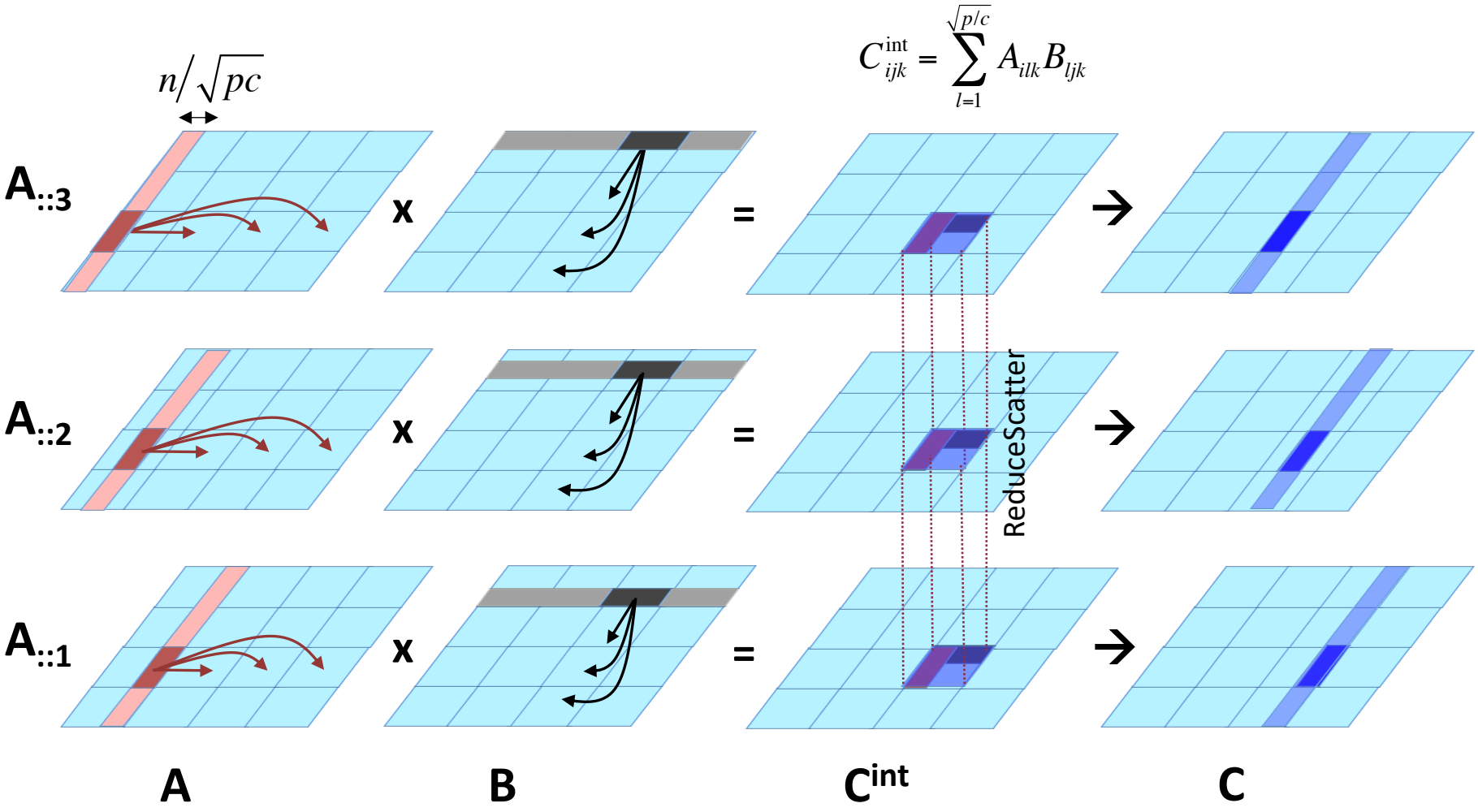


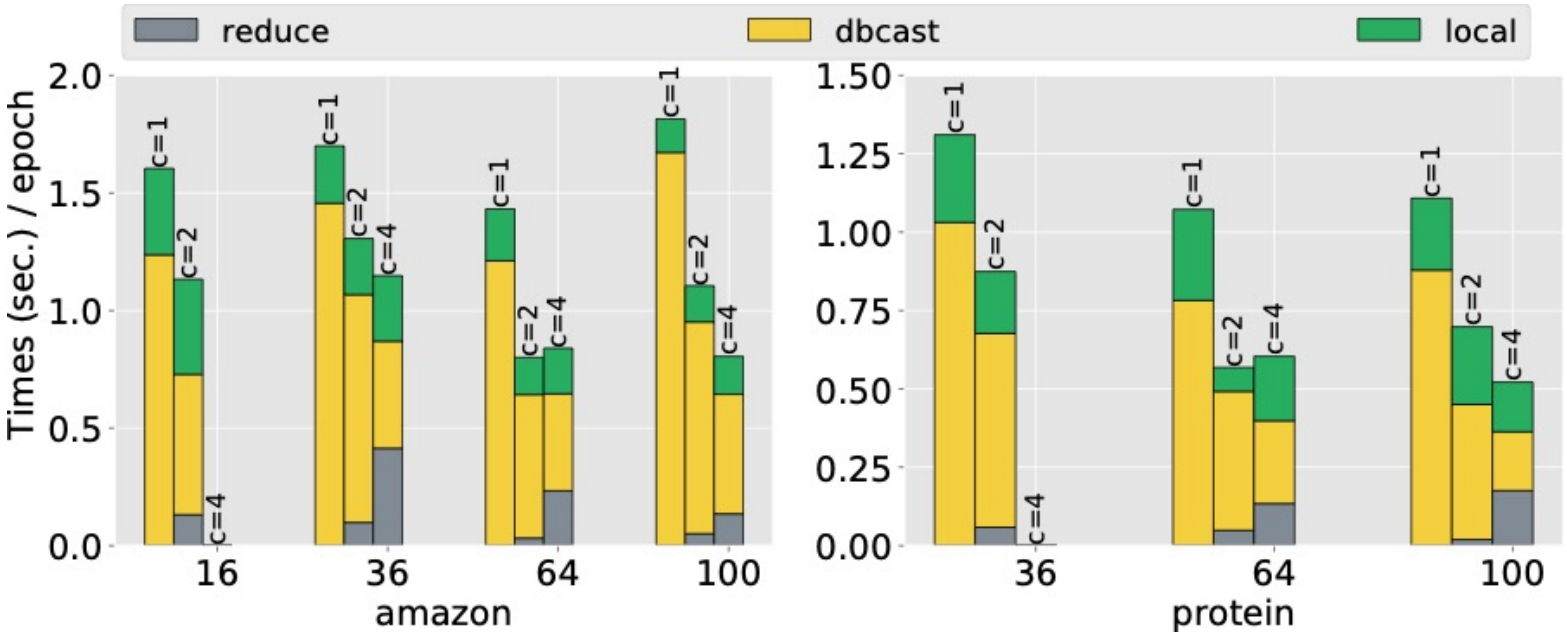
Illustration of the 3D algorithm on a $\sqrt{p/c} \times \sqrt{p/c} \times c$ process grid

Communication analysis

CAGNET Cost Analyses (per process)			
Algorithm	Latency	Bandwidth	Memory
1D	$\lg P + 2P$	$2nf + f^2$	$\frac{nnz(\mathbf{A})+nfL}{P}$
1.5D	$2\frac{P}{c^2}\lg\frac{P}{c^2}$	$\frac{2nf}{c} + \frac{2nfc}{P}$	$\frac{nnz(\mathbf{A})+nfL}{P} + \frac{nfc}{P}$
2D	$5\sqrt{P} + 3\lg P$	$\frac{8nf}{\sqrt{P}} + \frac{2nnz(\mathbf{A})}{\sqrt{P}}$	$\frac{nnz(\mathbf{A})+nfL}{P}$
3D	$4P^{1/3}$	$\frac{2nnz(\mathbf{A})}{P^{2/3}} + \frac{12nf}{P^{2/3}}$	$\frac{nnz(\mathbf{A})+nfL}{P} + \frac{nfc}{P}$

Symbols and Notations	
Symbol	Description
$\mathbf{A}$	Modified adjacency matrix of graph $(n \times n)$
$\mathbf{H}^l$	Embedding matrix in layer l $(n \times f)$
$\mathbf{W}^l$	Weight matrix in layer l $(f \times f)$
$\mathbf{Y}^l$	Matrix form of $\frac{\partial \mathcal{L}}{\partial W^l_{ij}}$ $(f \times f)$
$\mathbf{Z}^l$	Input matrix to activation function $(n \times f)$
$\mathbf{G}^l$	Matrix form of $\frac{\partial \mathcal{L}}{\partial Z^l_{ij}}$ $(n \times f)$
σ	Activation function
f	Length of feature vector per vertex
f_u	Feature vector for vertex u
L	Total layers in GNN
P	Total number of processes
α	Latency
β	Reciprocal bandwidth

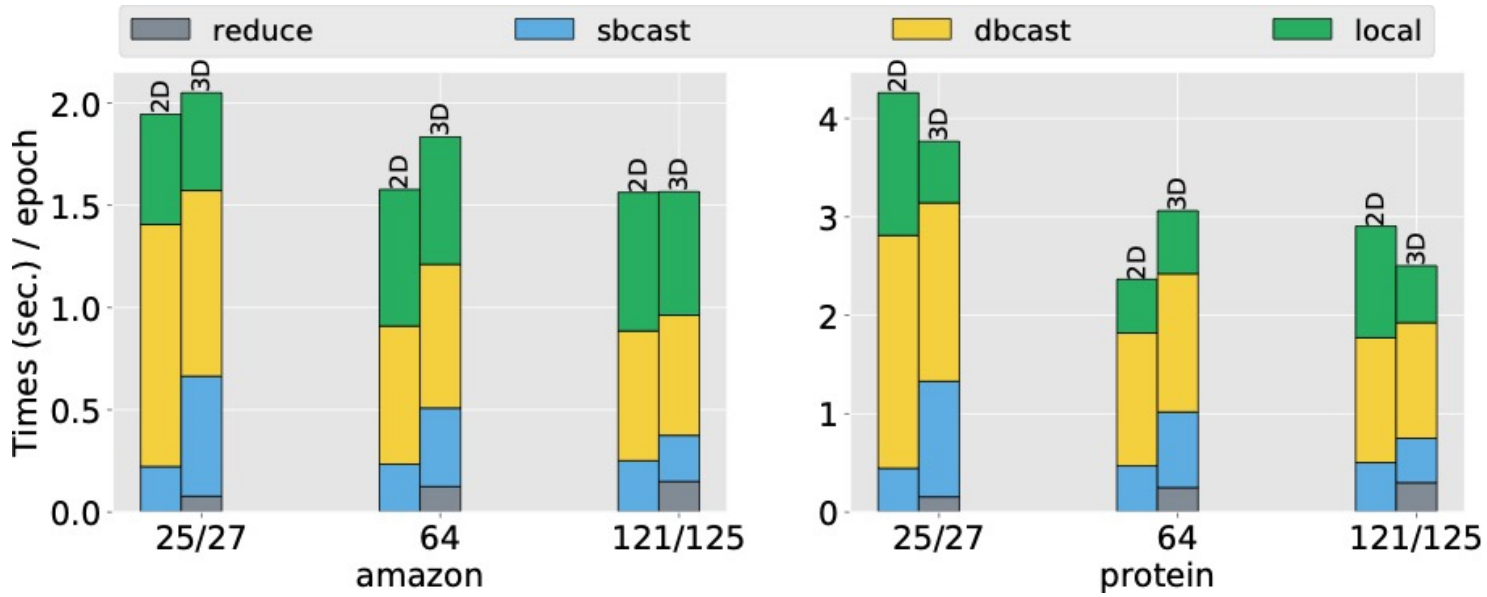
Communication avoidance (CA) in GNN Training



- Scales with both P (GPUs – x axis) and c (replication layers in CA algorithms)
- This is 1 GPU/node on Summit (all GPUs per node results in paper)
- Expect to scale with all GPUs / node with future architectures (e.g. Perlmutter)

Alok Tripathy, Katherine Yelick, Aydın Buluç. Reducing Communication in Graph Neural Network Training. SC'20

2D vs. 3D performance



- 64 hidden-layer activations
- Communication scales with P, consistent with analysis
- Computation scales less well → explained in paper

Alok Tripathy, Katherine Yelick, Aydın Buluç. Reducing Communication in Graph Neural Network Training. SC'20

Parallel GNN training conclusions

- Graph representation learning is transforming science
  - » Lots of deep learning problems on graphs
- Can solve DL on graphs with GNNs
  - » But must distribute training
- Alok's work
  - » Can formulate GCN training as SpMM
  - » Distribute GCN training with distributed SpMM
  - » Code: https://github.com/PASSIONLab/CAGNET
- Future work
  - » Distributed sampling for mini-batch training [next slide]
  - » Beyond graph convolutions [after next slide]