Dissertation Defense

Distributed Sparse Computing and Communication for Big Graph Analytics and Deep Learning

by

Mohammad Hasanzadeh Mofrad

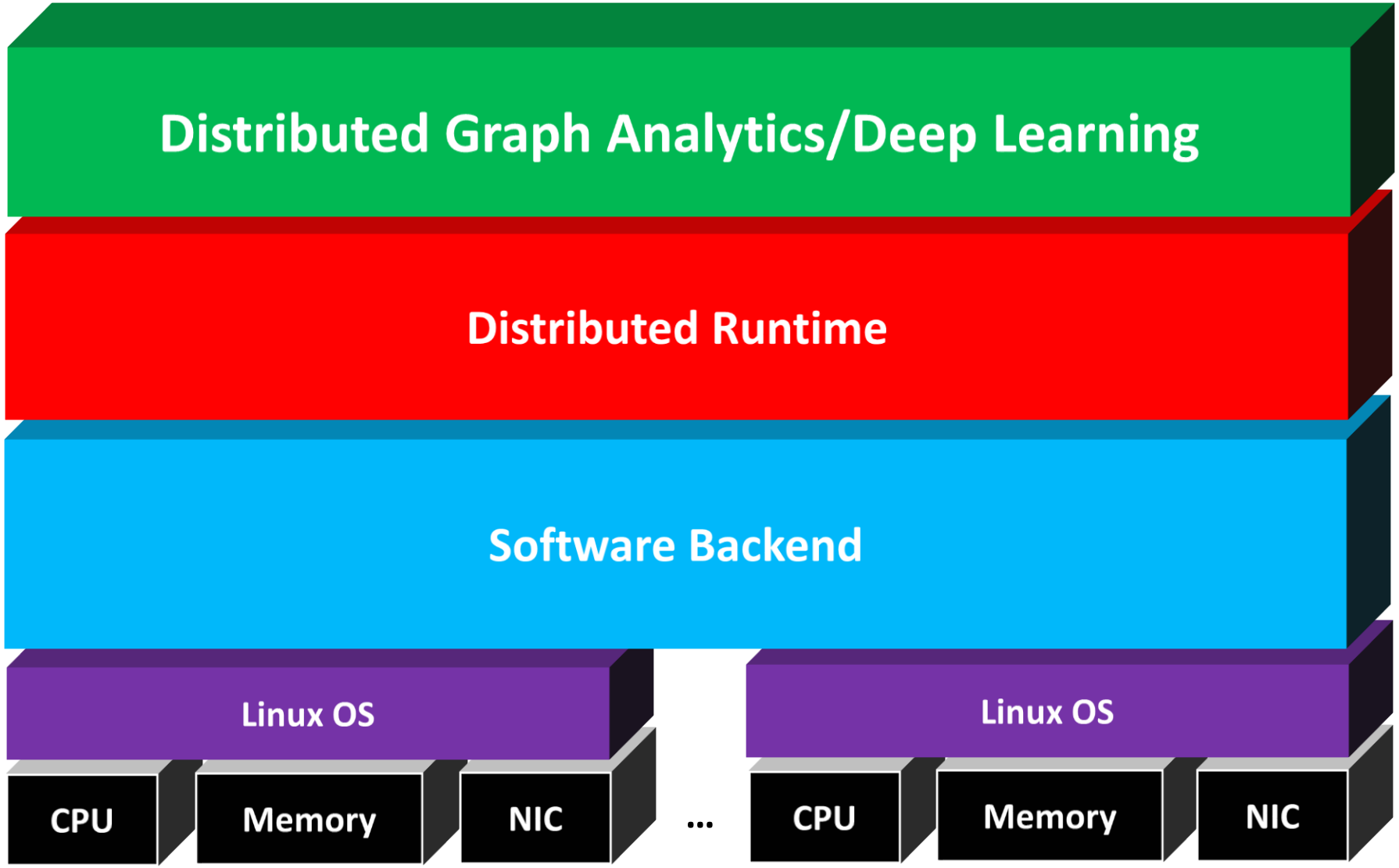
Date: Friday, October 30, 2020

Time: 9:30 AM EST

Committee:

Dr. Rami Melhem (advisor), Univ. of Pittsburgh
Dr. Alexandros Labrinidis, Univ. of Pittsburgh
Dr. John Lange, Univ. of Pittsburgh
Dr. Balaji Palanisamy, Univ. of Pittsburgh
Dr. Mohammad Hammoud, CMU-Qatar

Overview: Distribute Sparse Computing and Communication



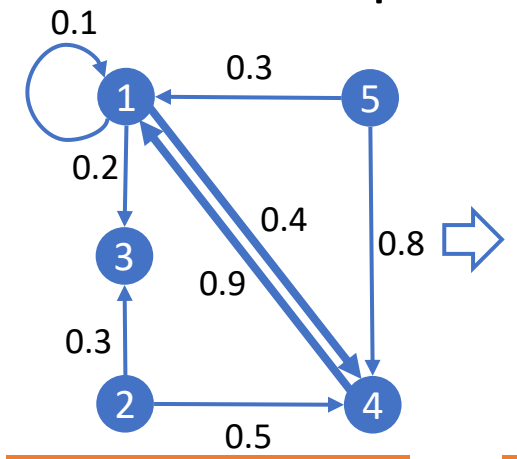
Motivation: Graph Analytics

- **Big data** is a multi-billion-dollar industry
  - $230 Billion by 2025 www.prnewswire.com
- **Relational data** represented by **graphs** is dominating the big data industry

- Graphs are ubiquitous
  - **Google** PageRank
    - Internet graph
  - **Uber** routing's engine
    - Commute graph
  - **Amazon** item recommendation
    - Bipartite items/users' graph
  - **Facebook** friend suggestion
    - Relationship graph
- Majority of these graphs are sparse

Background: Duality Between Graphs & Sparse Matrices

- The relational data represented by graphs is often **hyper-sparse**
- Linear algebra primitives backed by sparse matrix formats are new alternatives to their graph-based versions



Graph **G** (*n* vertices)

Src	Dst	Wgt
1	1	0.1
1	3	0.2
1	4	0.4
2	3	0.3
2	4	0.5
4	1	0.9
5	1	0.3
5	4	0.8

Adjacency List

	0	1	2	3	4	5
0						
1		.1		.2	.4	
2				.3	.5	
3						
4		.9				
5		.3			.8	

Adjacency Matrix $\mathbf{A}_{nxn}$

	Graph Theory	**Linear Algebra**
Representation	Graph	Adjacency Matrix
Data structure	Adjacency List	CSC[1], CSR[2], DCSC[3], DCSR[4]
Computing Model	Fan-in/ Fan-out Operations	GAS[5]
Primitive		**SpMV[6]**
Scalability	Graph Partitioning	Sparse Matrix Partitioning

1. Compressed Sparse Column (CSC)
2. Compressed Sparse Row (CSR)
3. Doubly Compressed Sparse Column (DCSC)
4. Doubly Compressed Sparse Column (DCSR)
5. Gather, Apply, and Scatter (GAS) abstraction
6. Sparse Matrix – Vector (SpMV)

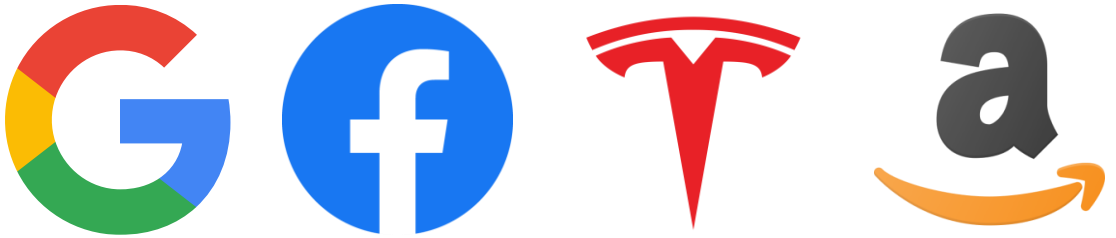
State-of-the-art Comparison: Graph Analytics

System	Computing Model	Platform	Parallelism Model	Computation & Communication	Matrix Compression	Authors
GraphPad	Linear Algebra	HPC/Cloud	MPI+OpenMP		DCSC	2016, Intel
Gemini	Frontier-based	HPC	MPI+OpenMP	NUMA-aware CPU	CSR/CSC (Dual)	2016, Tsinghua University
CombBLAS	Linear Algebra	HPC	MPI+OpenMP		DCSC	2018, LBNL
LA3	Linear Algebra	Cloud	MPI+OpenMP	Optimized for cloud communication	DCSC	2018, CMUQ
LAGraph	Graph Theory	HPC	OpenMP		CSR	2020, Texas A&M

- State-of-the-art big graph analytics systems are not 100% mature yet!

Motivation: Deep Learning

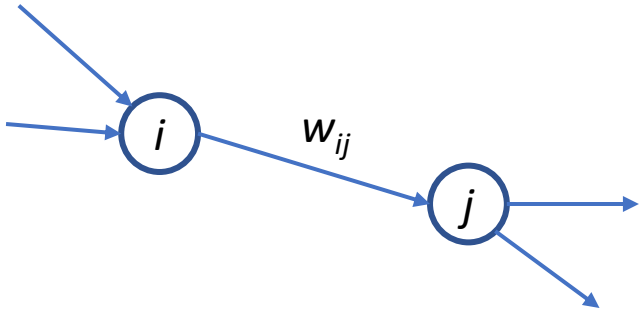
- **Deep Neural Networks (DNNs)** are pervasive
  - Speech processing
  - Post Translation
  - Autonomous driving
  - Knowledge Discovery

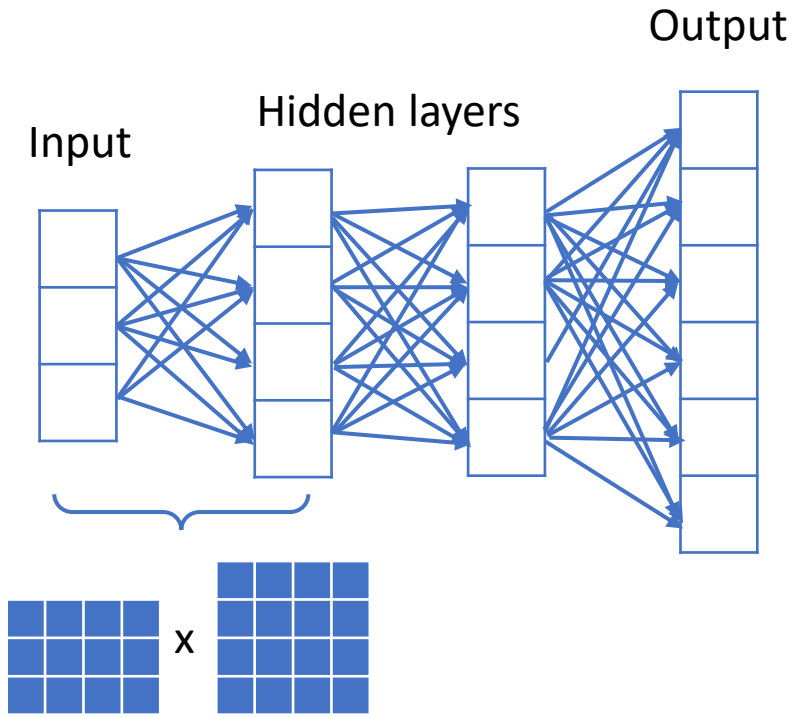


- The core kernel behind inference/training of DNNs is **Dense matrix-matrix multiplication (**$C_{m \times n} = A_{m \times n} \times B_{n \times n}$**)**
- **Sparse DNNs** are new alternative to dense DNNs with
  - Less time and space complexities
  - Better or comparable accuracy
  - Avoids overfitting
- **Sparse DNNs** core kernel is **Sparse Matrix-matrix Multiplication (SpMM)**

Background: Duality Between Linear Algebra & Deep Learning

- Neural networks connections
  - Triplet format graph theory: (i, j, w_{ij})
- Primitive: Matrix-matrix multiplication
  - $C = A \times B$
- Iterative matrix-matrix multiplication for DNN
  - $C_{l+1} = h(A_l \times B_l + b)$
- For Sparse neural networks, the primitive is
  - **SpMM** (Sparse Matrix-matrix multiplication)





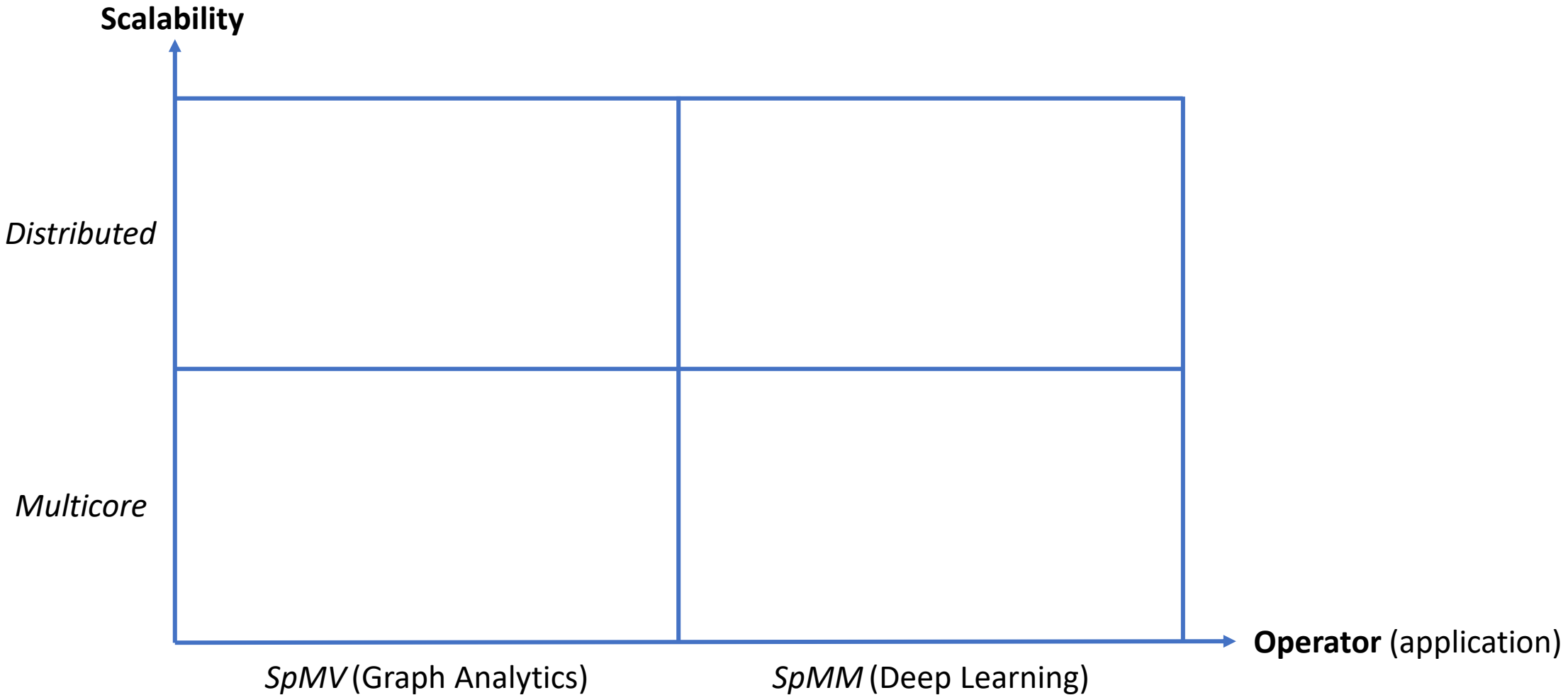
State-of-the-art Comparison: Deep Learning

System	Computing Model	Platform	Parallelism Model	Computation & Communication	Matrix Compression	Authors
TensorFlow	Linear Algebra	Commodity/ HPC/ Cloud	Data parallelism (Keras + CPU/GPU/TPU)	Horovod	Dense	2015 – present, Google
PyTorch	Linear Algebra	Commodity/ HPC	Data parallelism (MPI/ Gloo + OpenMP/ CUDA)	Horovod	Dense	2016 – present, Facebook
MXNet	Linear Algebra	Commodity/ HPC	Data parallelism (MPI4Py/ Gloo + OpenMP/ CUDA)	Horovod/ TCP/IP socket	Dense	2020 – present, Amazon
DeepSpeed	Linear Algebra	Commodity/ HPC/ Cloud	Data parallelism (MPI/ NCCL + X/ CUDA)	Communication Overlapping	Sparse	2019 – present, Microsoft

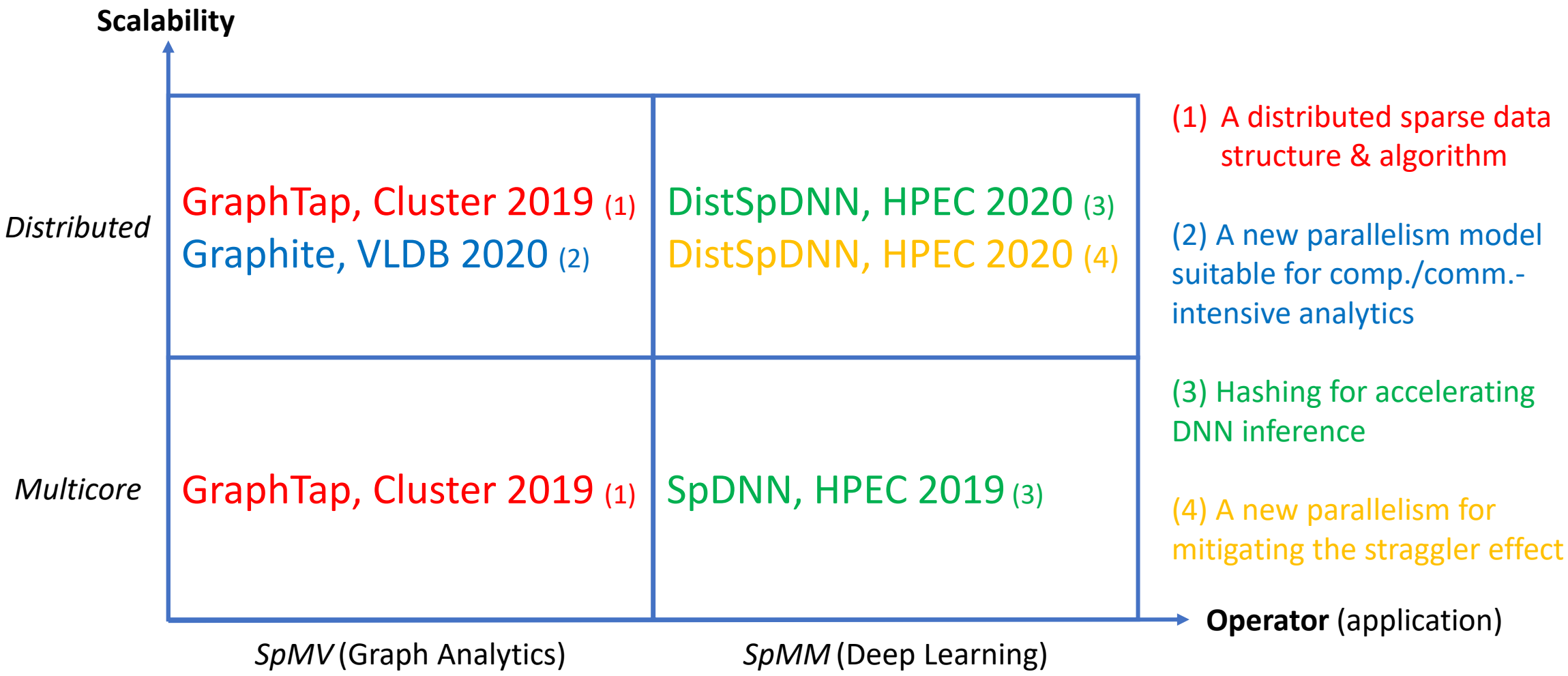
- Deep Learning frameworks are still evolving …

1. NVIDIA Collective Communications Library (NCCL)

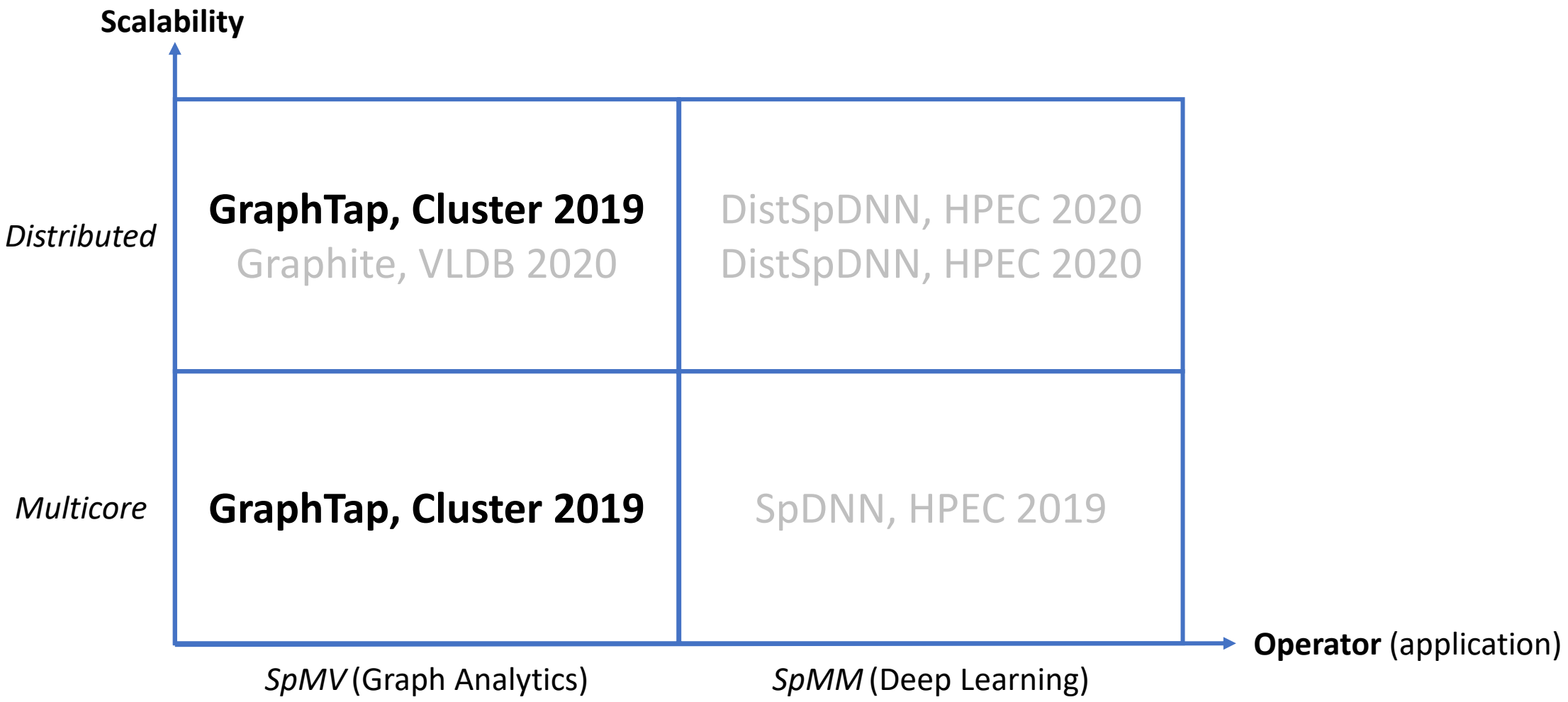
Contributions Overview



Contributions Overview

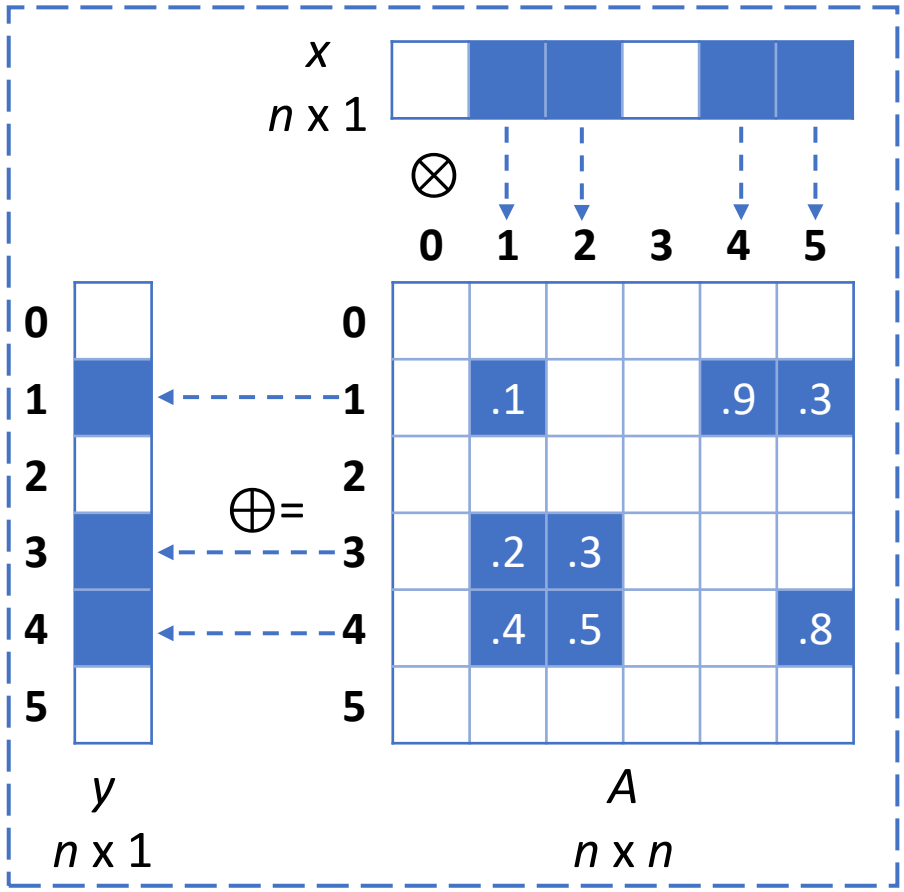


Contributions Overview

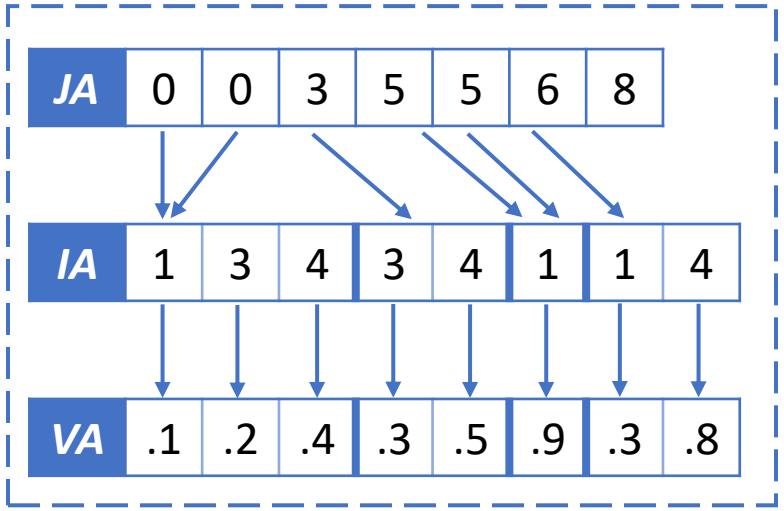


Compressed Sparse Column (CSC)

CSC space requirement is ***n* + 2*nnz* + 1** where *nnz* is the number of nonzero entries
+ Sequential column-major access (No indirection for SpMV)
- Length of *JA* and, *x* and *y* vectors equals *n*



CSC SpMV Diagram ($y = A \oplus . \otimes x$)

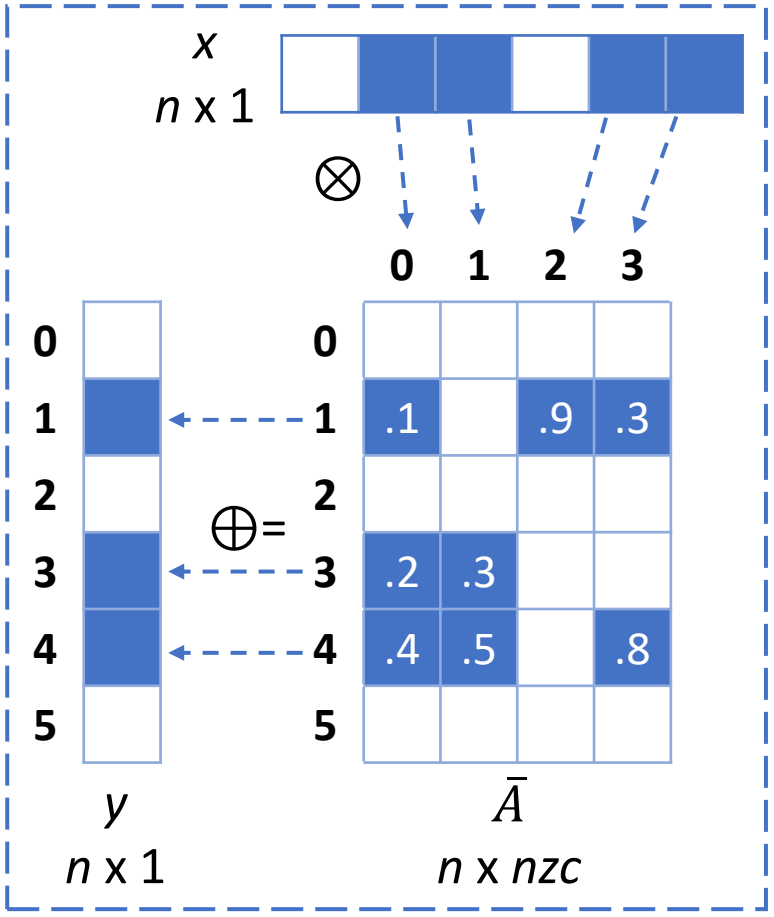


CSC Data Structures

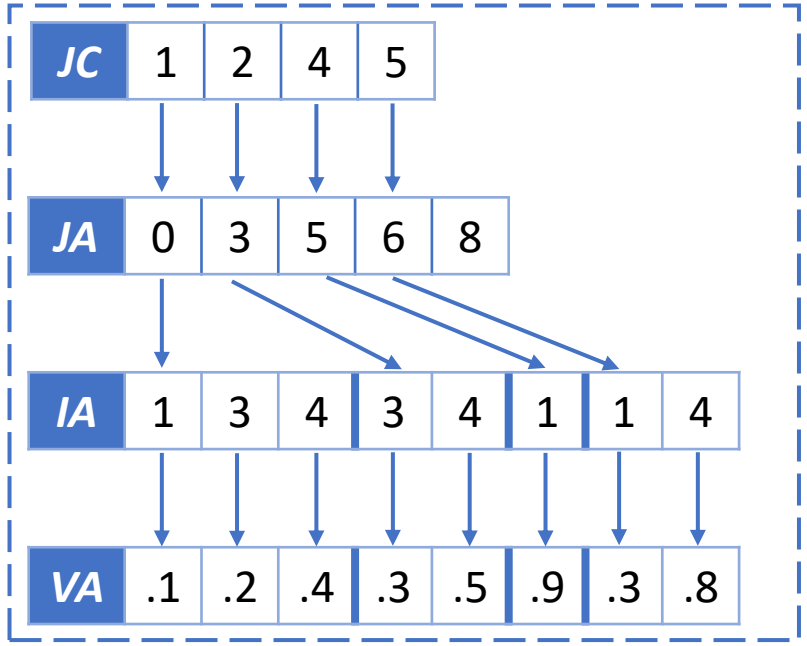
Doubly Compressed Sparse Column (DCSC)

CSC space requirement is ***n* + 2*nnz* + 1** where *nnz* is the number of nonzero entries
+ Sequential column-major access (No indirection for SpMV)
- Length of *JA* and, *x* and *y* vectors equals *n*

DCSC Space requirement is **2*nzc* + 2*nnz* + 1** where *nzc* is the number of nonzero columns
+ One indirection for SpMV (*n* → *nzc*)
+ Length of *JA* equals *nzc*
- Length of *x* and *y* vectors equals *n*



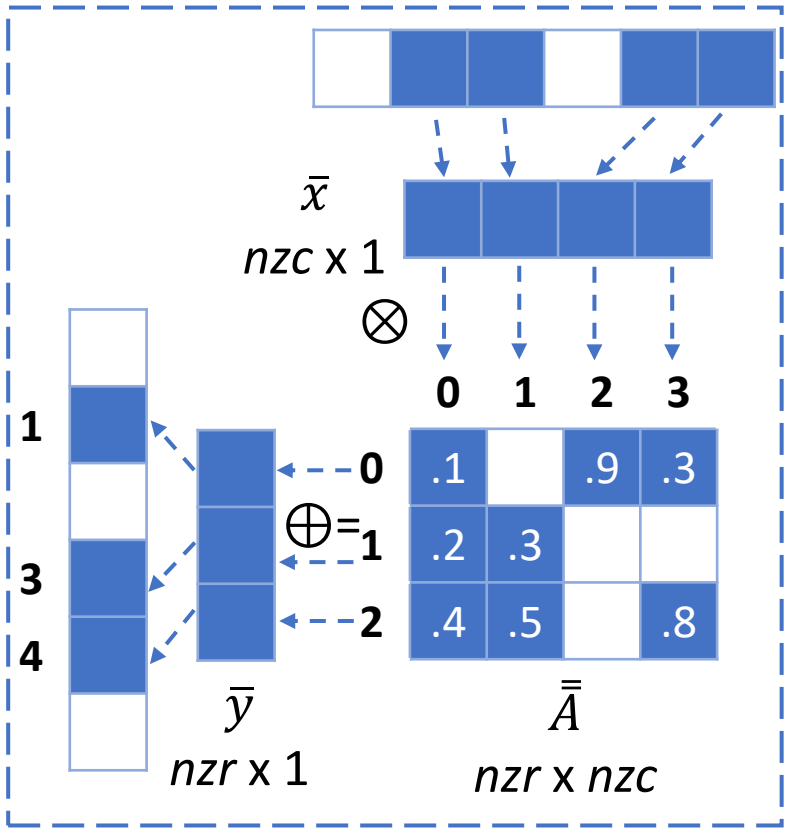
DCSC SpMV Diagram ($y = \bar{A} \oplus . \otimes x$)



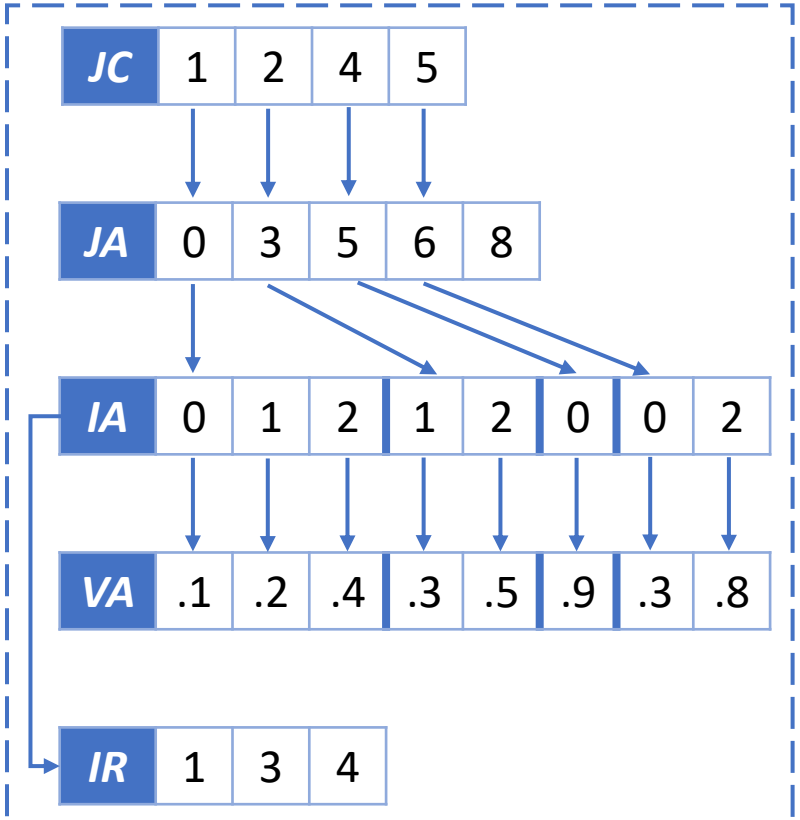
DCSC Data Structures

Triply Compressed Sparse Column (TCSC)

CSC space requirement is ***n* + 2*nnz* + 1** where *nnz* is the number of nonzero entries
+ Sequential column-major access (No indirection for SpMV)
- Length of *JA* and, *x* and *y* vectors equals *n*



TCSC SpMSpV² Diagram ($\bar{y} = \bar{\bar{A}} \oplus . \otimes \bar{x}$)



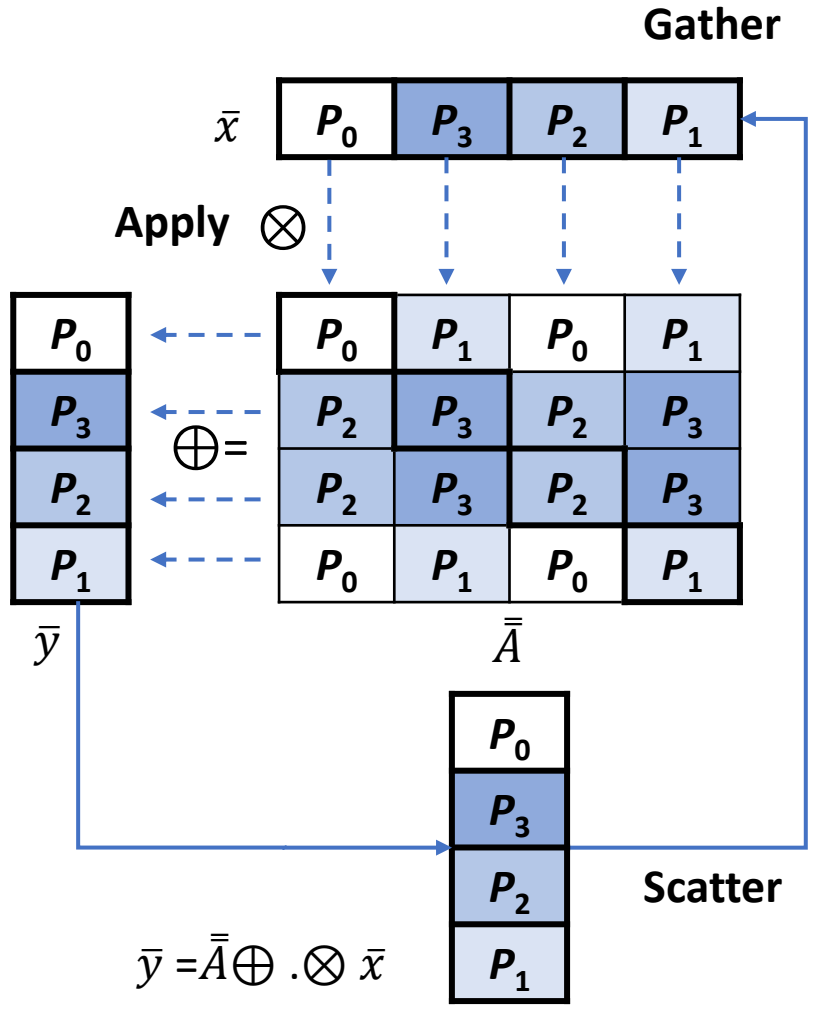
TCSC Data Structures

DCSC Space requirement is **2*nzc* + 2*nnz* + 1** where *nzc* is the number of nonzero columns
+ One indirection for SpMV (*n* → *nzc*)
+ Length of *JA* equals *nzc*
- Length of *x* and *y* vectors equals *n*

TCSC Space requirement is **2*nzc*+ *nzr*+ 2*nnz*+1** where *nzr* is the number of nonzero rows
+ No indirection for SpMSpV²
+ Length of *JA* equals *nzc*
+ Length of *x* and *y* vectors equals *nzc* and *nzr*

GraphTap: Distributed SpMSpV^2 Using TCSC for Graph Analytics

- **Graph computing model:** GAS
  - Gather (input vector), Apply (SpMspV^2), and Scatter (output vector)
- **Scalability**: 2D matrix partitioning
  - Tile size translates into computation volume
  - Vector size translates into communication volume
- **Compression**: TCSC
  - TCSC's less indirections means less computation time
  - TCSC's smaller vectors means less accumulation time
- **Communication**:
  - TCSC's smaller vectors means less communication time



Experiments

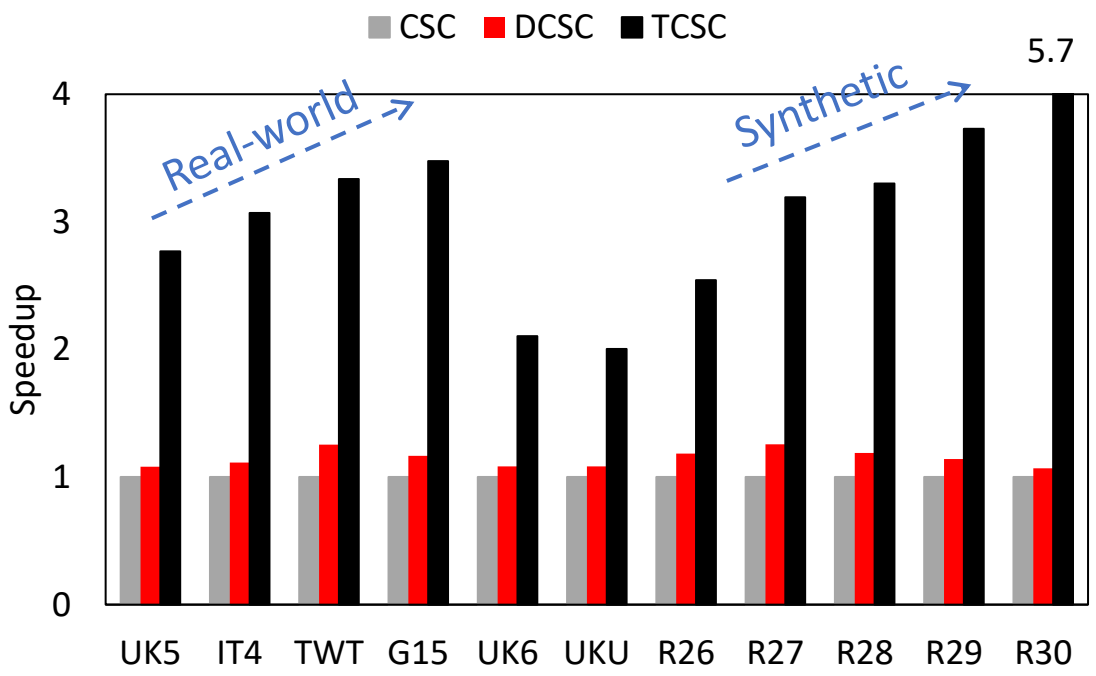
Graph Processing Systems		
System	**Compression**	**Primitive**
Graphite	TCSC	SpMSpV^2 GAS
LA3 (VLDB, 2018)	DCSC	SpMV GAS
GraphPad (IPDPS, 2016)	CSC	SpMV GAS

Node Specification (32)	
CPU	28-core @ 2.6 GHZ
Memory	192 GB
OS	Linux
MPI	Intel
Network	Intel Omni-path Fabric

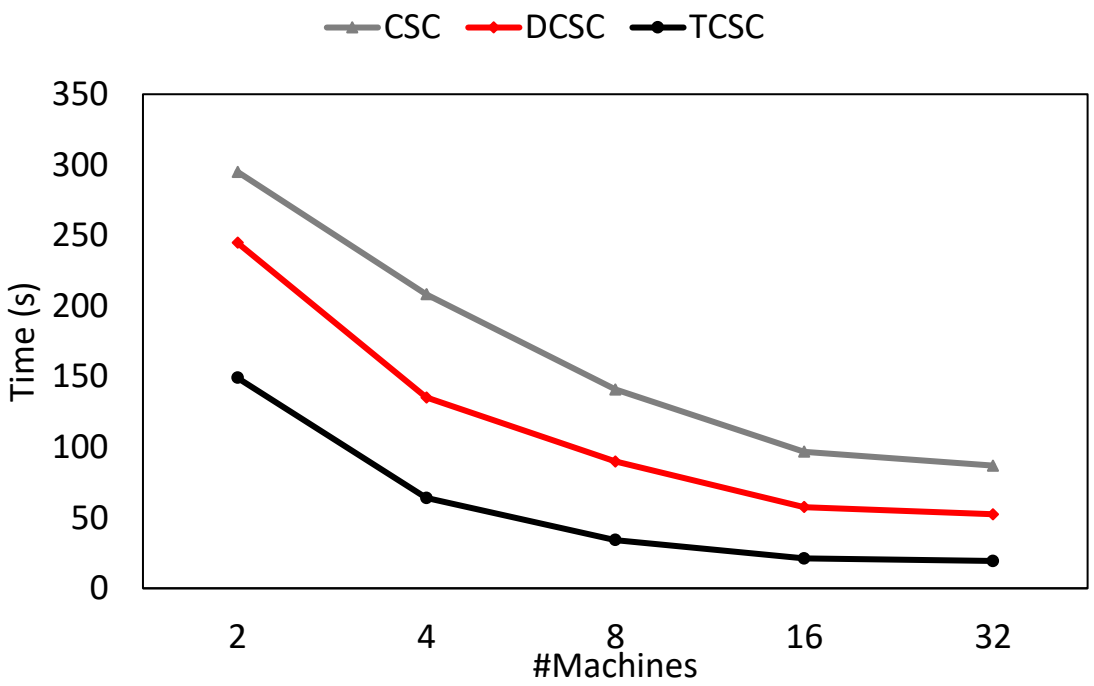
Graph Applications
PageRank (PR)
Single Source Shortest Path (SSSP)
Breadth First Search (BFS)
Connected Component (CC)

Dataset			
Graph	**$\|V\|$**	**$\|E\|$**	**Type**
UK'05 (UK5)	39.4 M	0.93 B	Web
IT'04 (IT4)	41.2 M	1.15 B	Web
Twitter (TWT)	41.6 M	1.46 B	Social
GSH'15 (G15)	68.6 M	1.8 B	Web
UK'06 (UK6)	80.6 M	2.48 B	Web
UK Union (UKU)	133 M	5.5 B	Web
Rmat26 (R26)	67.1 M	1.07 B	Synthetic
Rmat27 (R27)	134 M	2.14 B	Synthetic
Rmat28 (R28)	268 M	4.29 B	Synthetic
Rmat29 (R29)	536 M	8.58 B	Synthetic
Rmat30 (R30)	**1.07 B**	**17.1 B**	**Synthetic**

Experiments: CSC, DCSC, and TCSC Comparison



PR weak scaling (up to 6x speedup)

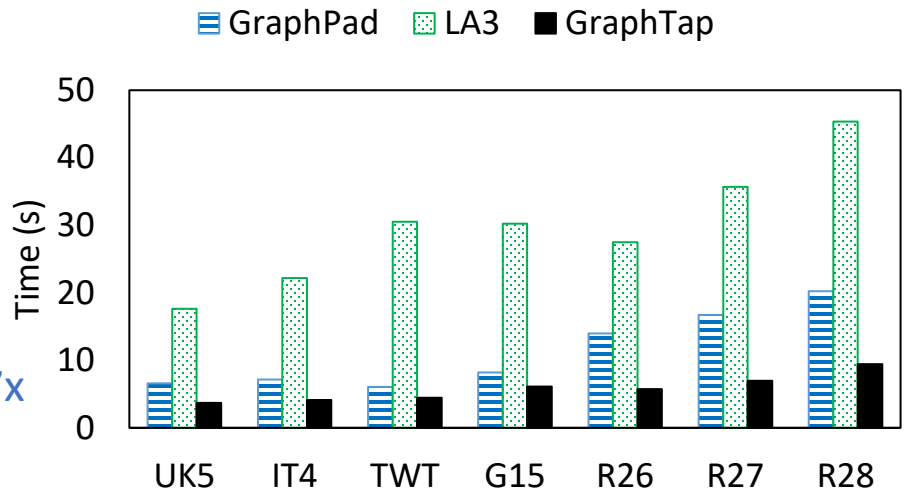


PR Cluster scaling

Experiments: GraphTap, GraphPad, and LA3 Comparison (weak scaling)

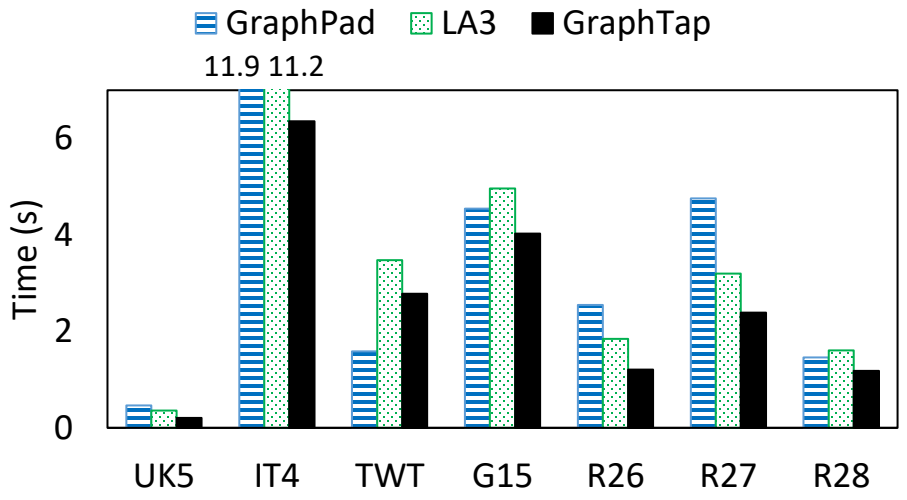
PR

Up to 7x



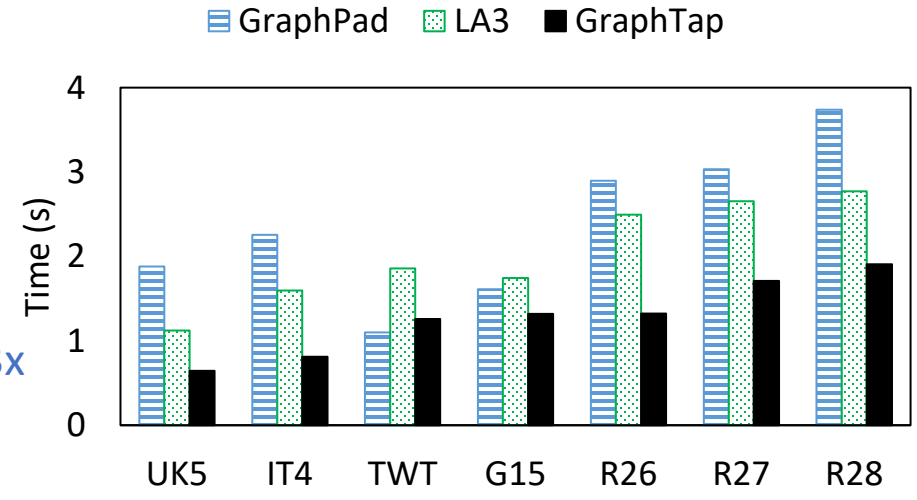
SSSP

Up to 3x



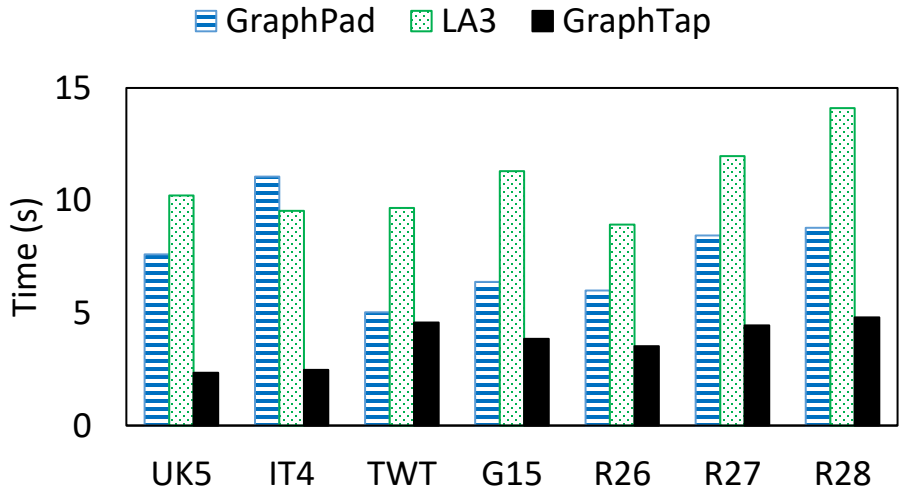
BFS

Up to 3x

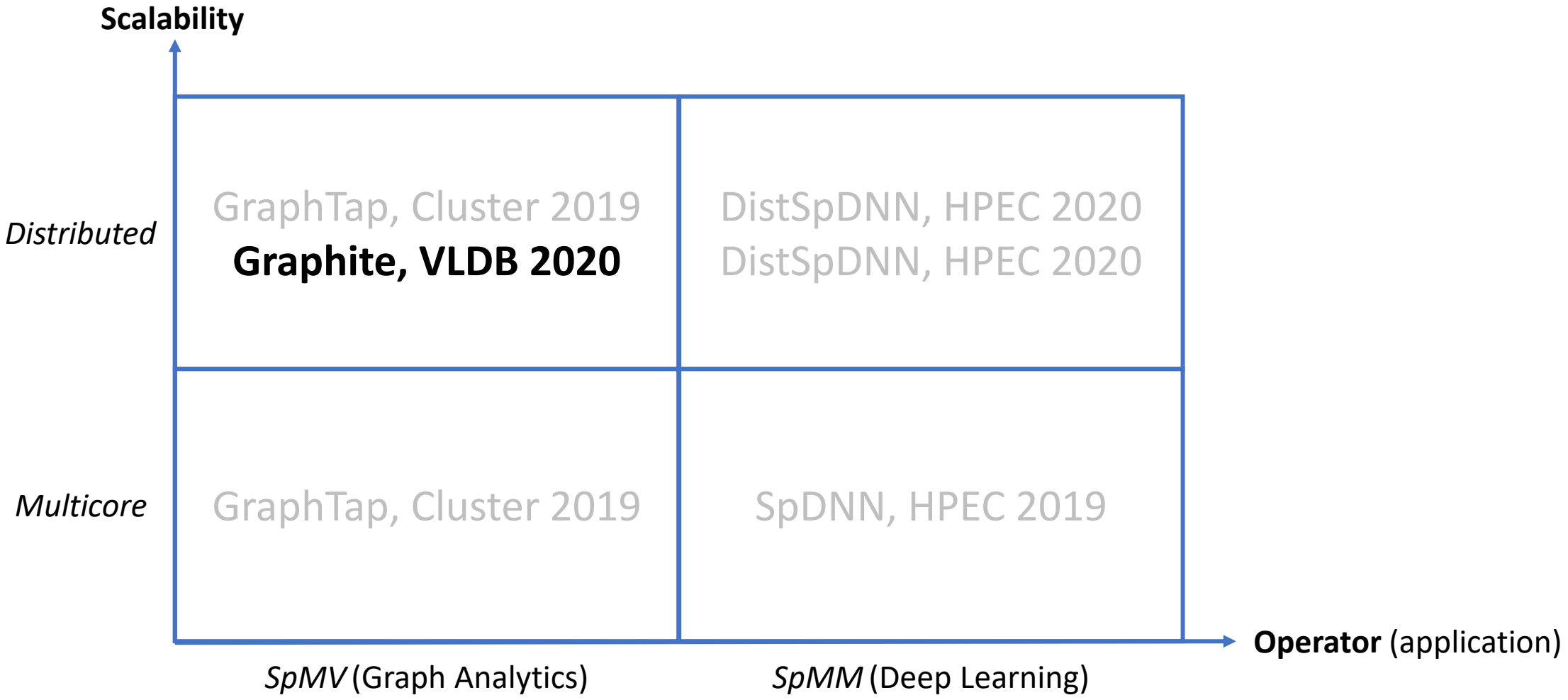


CC

Up to 4.5x

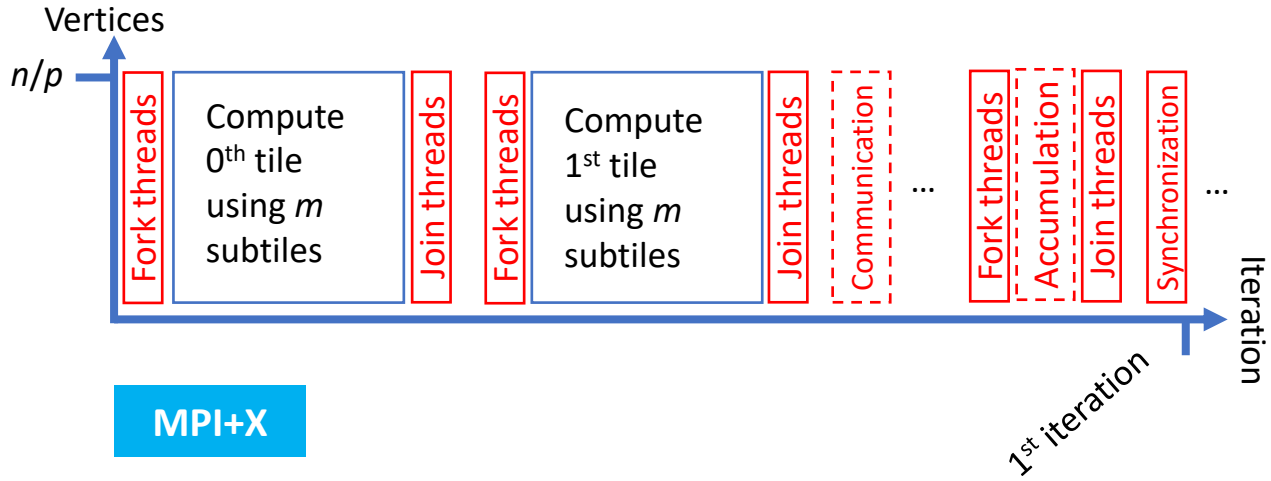


Contributions Overview

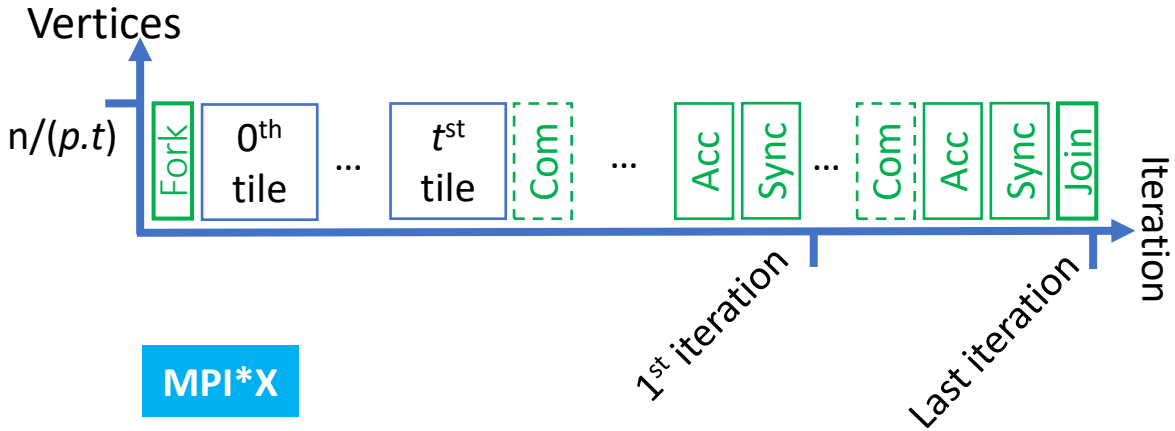


MPI+X Parallelism Model vs. the New MPI*X Parallelism Model

- MPI+X uses process-based partitioning and hence
  - When computing each tile, multiple threads are forked & joined
  - Only MPI processes are the communication endpoints
  - Synchronization points are designed for MPI processes



- MPI*X uses thread-based partitioning and so
  - Threads stay alive
  - Communication is done by threads (better overlapping)
  - Synchronization is done by threads

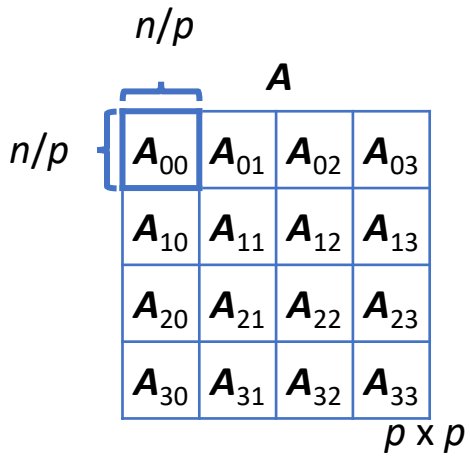


The Process-based 2D Matrix Partitioning and Placement for scalability (e.g., p=4)

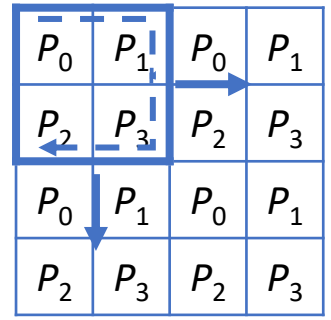
- **Processed-based 2D partitioning**: Given p processes, a matrix is partitioned into a p x p grid of tiles
  - Each process **computes** p tiles

- **2D cyclic placement (GraphPad, IPDPS 16)**: Given p processes, $\sqrt{p}$ processes are placed in each row/column group
  - Process **communication** is not part of the placement

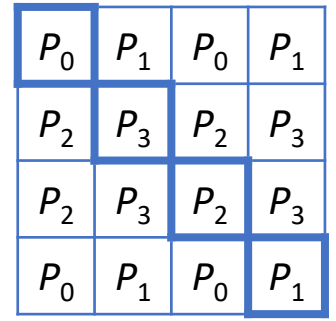
- **2D Staggered partitioning (LA3, VLDB 18)**: Like 2D cyclic, however, unique processes are placed in diagonal tiles
  - Process **communication** is democratized by diagonal processes (O(2p)+O(p^2) time)



2D-process-based partitioning



2D-Cyclic Placement

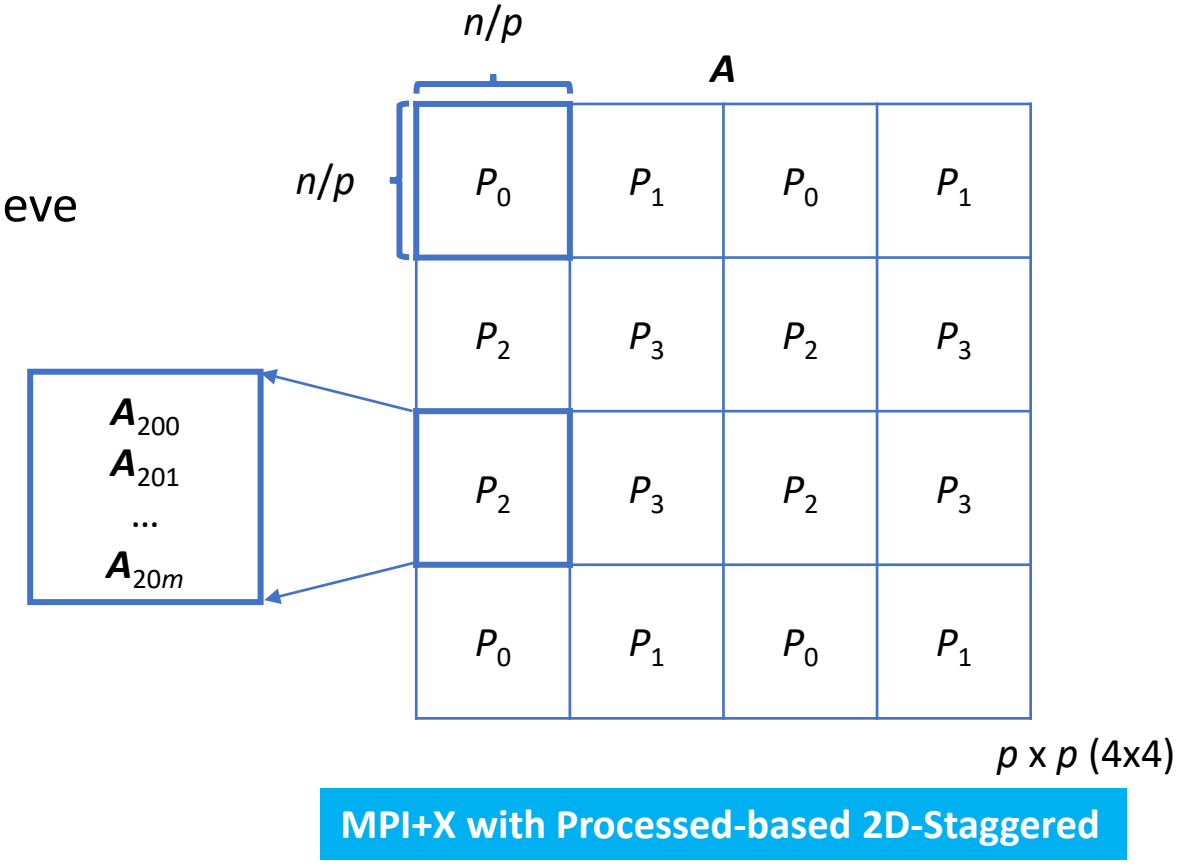


2D-Staggered Placement

MPI+X Parallelism Model for Multicore HPC Systems (e.g., *p*=4)

- **MPI+X parallelism model** utilizes
  - MPI to achieve **horizontal scaling**
    - One MPI process per machine
  - A threading library (X), e.g, OpenMP, or Pthread to achieve **vertical scaling**

- **MPI+X** uses process-based partitioning & placement

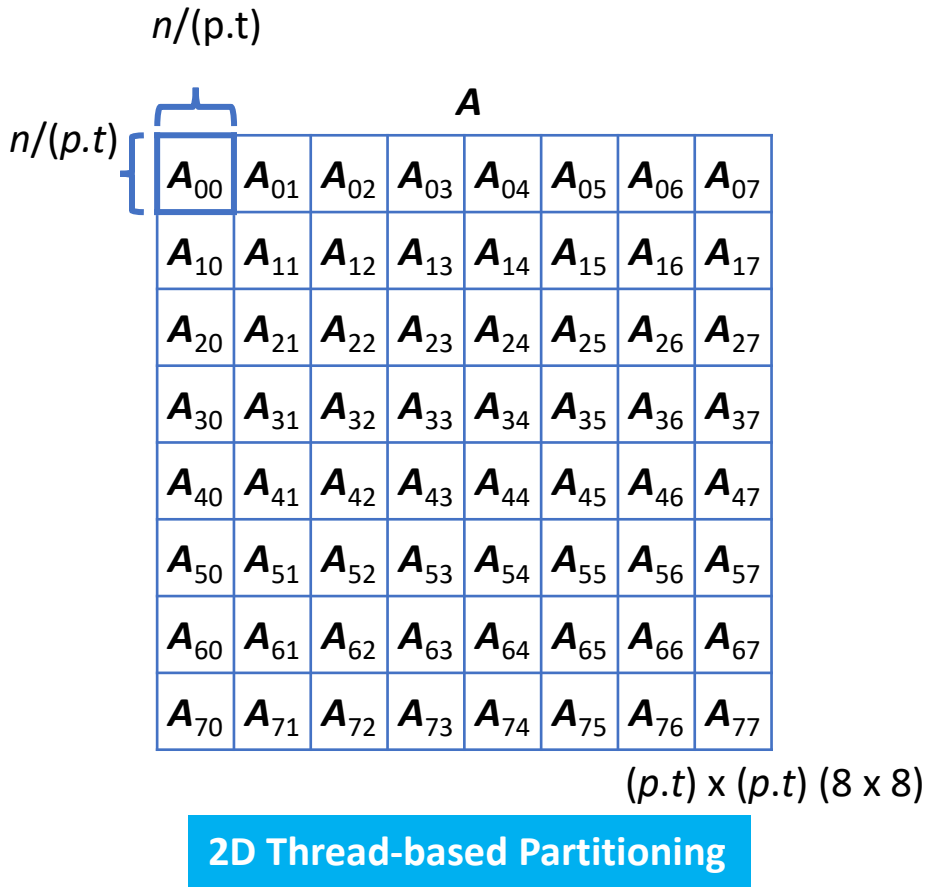


MPI+X with Processed-based 2D-Staggered

m is the number of sub-partitions

The New Thread-based 2D Matrix Partitioning and Placement for scalability (e.g., p=4, t=2)

- **Thread-based 2D partitioning**: Given p processes & t threads, a matrix is partitioned into a ($p.t$)x($p.t$) grid of tiles
  - $p.t$ global threads
  - Each global thread **computes** t tiles



The New Thread-based 2D Matrix Partitioning and Placement for scalability (e.g., p=4, t=2)

- **Thread-based 2D partitioning**: Given p processes & t threads, a matrix is partitioned into a ($p.t$)x($p.t$) grid of tiles
  - $p.t$ global threads
  - Each global thread **computes** t tiles

- **New Thread-based 2D Staggered placement (2DT-Staggereed)**:
  - Thread **communication** is democratized by diagonal threads (O(2$p.t$) time)

A

T_0	T_1	T_0	T_1	T_0	T_1	T_0	T_1
T_2	T_3	T_2	T_3	T_2	T_3	T_2	T_3
T_4	T_5	T_4	T_5	T_4	T_5	T_4	T_5
T_6	T_7	T_6	T_7	T_6	T_7	T_6	T_7
T_2	T_3	T_2	T_3	T_2	T_3	T_2	T_3
T_4	T_5	T_4	T_5	T_4	T_5	T_4	T_5
T_6	T_7	T_6	T_7	T_6	T_7	T_6	T_7
T_0	T_1	T_0	T_1	T_0	T_1	T_0	T_1

($p.t$) x ($p.t$) (8 x 8)

2D Thread-based Placement (Global Thread Ids)

The New MPI*X Parallelism Model (e.g., p=4 and t=2)

- **MPI*X parallelism model** utilizes
  - Threads to achieve **diagonal scaling**
    - t global threads are grouped together to form a process
  - Global thread ids are combined to form process and local thread assignments
  - One MPI process per CPU socket and one thread per core
    - NUMA-aware

A

P_0T_0	P_1T_0	P_0T_0	P_1T_0	P_0T_0	P_1T_0	P_0T_0	P_1T_0
P_2T_0	P_3T_0	P_2T_0	P_3T_0	P_2T_0	P_3T_0	P_2T_0	P_3T_0
P_0T_1	P_1T_1	P_0T_1	P_1T_1	P_0T_1	P_1T_1	P_0T_1	P_1T_1
P_2T_1	P_3T_1	P_2T_1	P_3T_1	P_2T_1	P_3T_1	P_2T_1	P_3T_1
P_2T_0	P_3T_0	P_2T_0	P_3T_0	P_2T_0	P_3T_0	P_2T_0	P_3T_0
P_0T_1	P_1T_1	P_0T_1	P_1T_1	P_0T_1	P_1T_1	P_0T_1	P_1T_1
P_2T_1	P_3T_1	P_2T_1	P_3T_1	P_2T_1	P_3T_1	P_2T_1	P_3T_1
P_0T_0	P_1T_0	P_0T_0	P_1T_0	P_0T_0	P_1T_0	P_0T_0	P_1T_0

($p.t$) x ($p.t$) (8 x 8)

2D Thread-based Placement (Local Thread Ids)

The **Graphite** HPC Graph Analytics System

- **MPI*X**
  - Thread-based partitioning & placement (O(2*pt*) time versus previous O(*p*^2))
  - Diagonal scaling
- **TCSC** (Triply Compressed Sparse Column) (GraphTap, Cluster 19)
- **NUMA-aware**
  - Computation (CPU and memory affinity)
  - Communication (MPI shared-memory transport)
- **Asynchronous** computation & communication
  - Broadcasting (log(*p*) time) when possible

Experiments

Graph Processing Systems

System	Parallelism	Unit	Partitioning	NUMA-aware	Compression	Primitive
Graphite	MPI*X	Threads	2DT-Staggered	CPU/Memory	TCSC	SpMSpV^2 GAS
LA3 (VLDB, 2018)	MPI+X	Processes	2D-Staggered	N/A	DCSC	SpMV GAS
GraphPad (IPDPS, 2016)	MPI+X	Processes	2D-Cyclic	N/A	CSC	SpMV GAS
Gemini (USENIX, 2016)	MPI+X	Processes	1D-Row	Memory	CSC/CSR	Push/Pull

Node Specification (20)

CPU	28-core @ 2.6 GHZ
Memory	192 GB
OS	Linux
MPI	Intel
Network	Intel Omni-path Fabric

Graph Applications

PageRank (PR)

Single Source Shortest Path (SSSP)

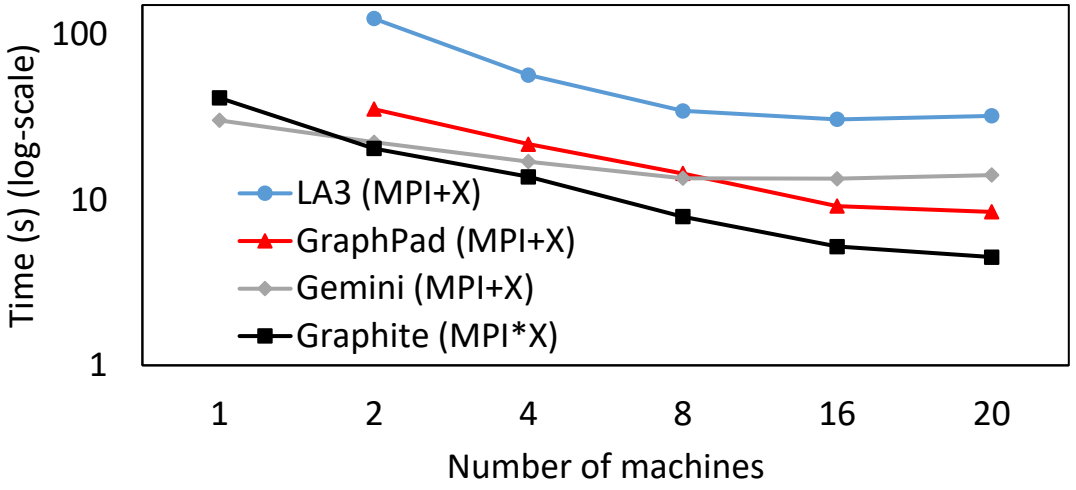
Breadth First Search (BFS)

Connected Component (CC)

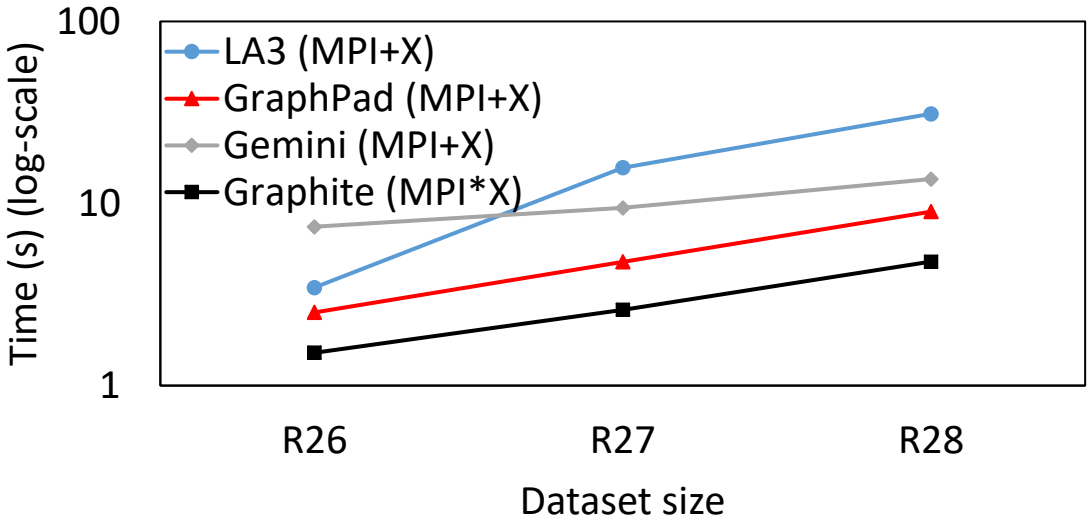
Datasets

Graph	$\|V\|$	$\|E\|$	Type
UK'05 (UK5)	39.4 M	0.93 B	Web
IT'04 (IT4)	41.2 M	1.15 B	Web
Twitter (TWT)	41.6 M	1.46 B	Social
GSH'15 (G15)	68.6 M	1.8 B	Web
UK'06 (UK6)	80.6 M	2.48 B	Web
UK Union (UKU)	133 M	5.5 B	Web
Rmat26 (R26)	67.1 M	1.07 B	Synthetic
Rmat27 (R27)	134 M	2.14 B	Synthetic
Rmat28 (R28)	268 M	4.29 B	Synthetic
Rmat29 (R29)	536 M	8.58 B	Synthetic

Experiments: MPI*X and MPI+X comparison (scalability)



PR Cluster Scaling

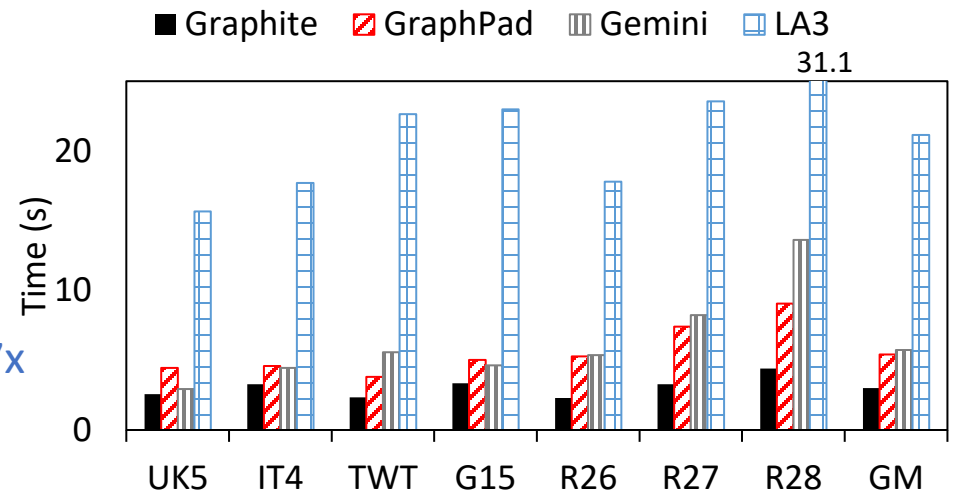


PR Data Scaling

Results (Weak Scaling)

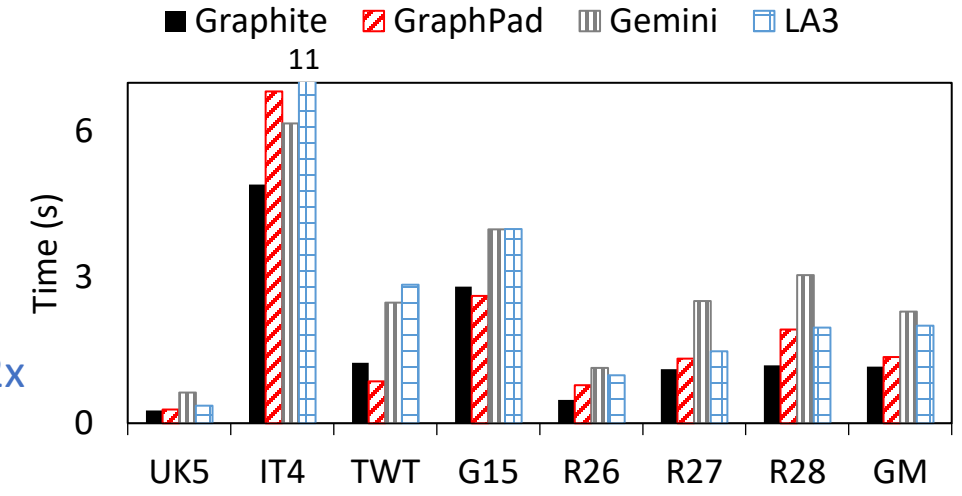
PR

Up to 7x



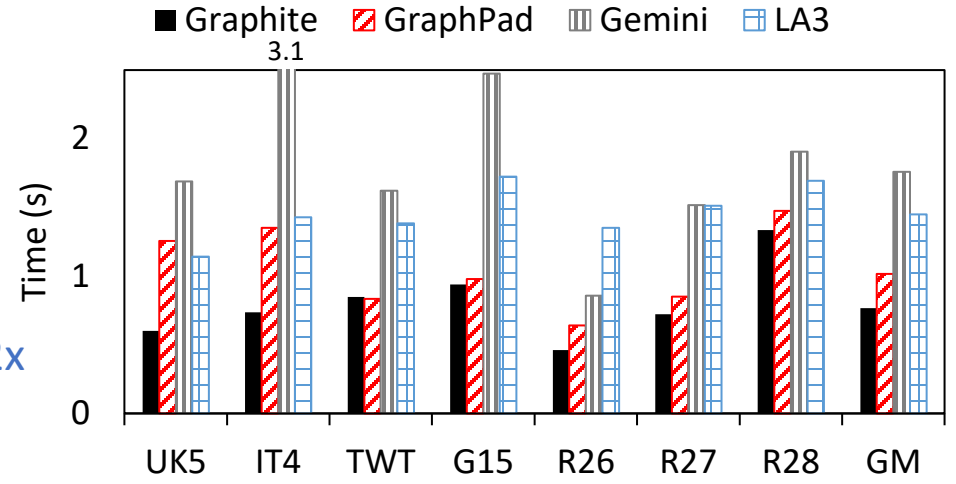
SSSP

Up to 2x



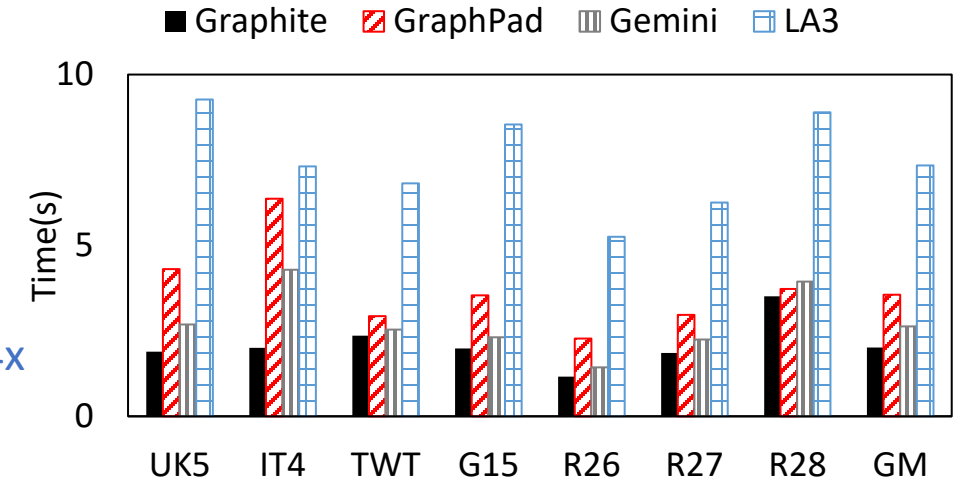
BFS

Up to 2x



CC

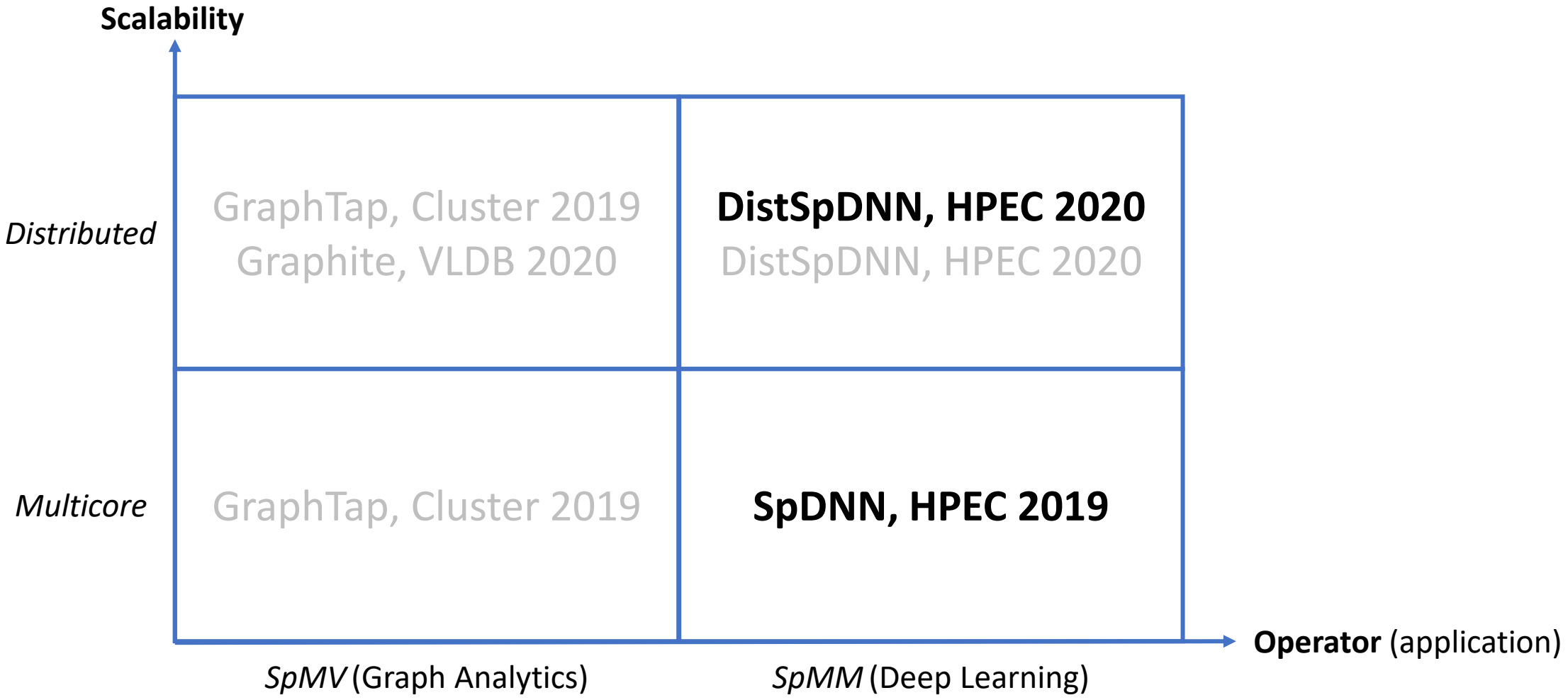
Up to 4x



GraphTap & Graphite vs. State-of-the-art

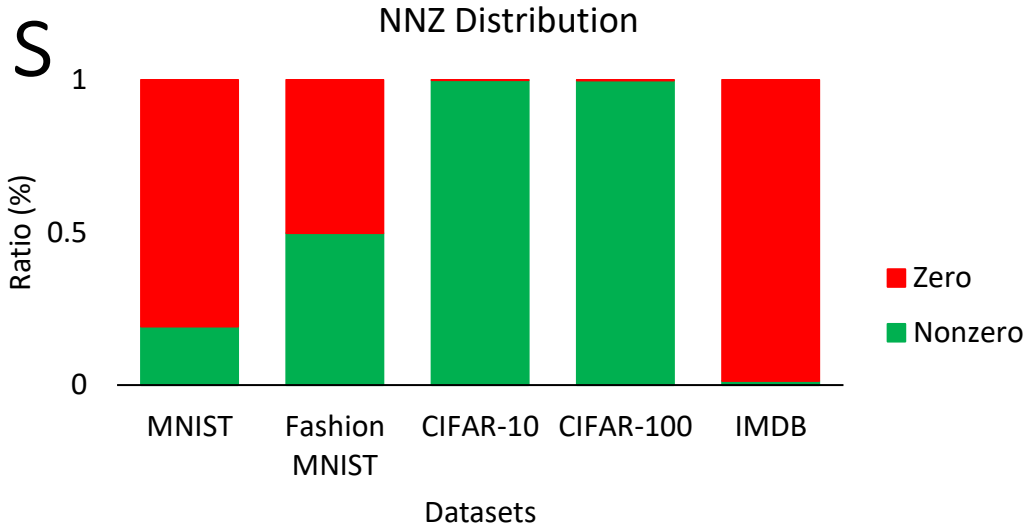
System	Computing Model	Platform	Parallelism Model	Computation & Communication	Matrix Compression	Authors
GraphPad	Linear Algebra	HPC/Cloud	MPI+OpenMP		DCSC[2]	2016, Intel
Gemini	Frontier-based	HPC	MPI+OpenMP	NUMA-aware CPU	CSR/CSC (Dual)	2016, Tsinghua University
CombBLAS	Linear Algebra	HPC	MPI+OpenMP		DCSC	2018, LBNL
LA3	Linear Algebra	Cloud	MPI+OpenMP	Optimized for cloud communication	DCSC	2018, CMUQ
LAGraph	Graph Theory	HPC	OpenMP		CSR	2020, Texas A&M
GraphTap	Linear Algebra	HPC	MPI		CSC/DCSC/ **TCSC**	2019, UPitt
Graphite	Linear Algebra	HPC	**MPI*X**	**Asynchronous Comm. Overlapping of Comm. NUMA-aware**	**TCSC**	2020, UPitt

Contributions Overview

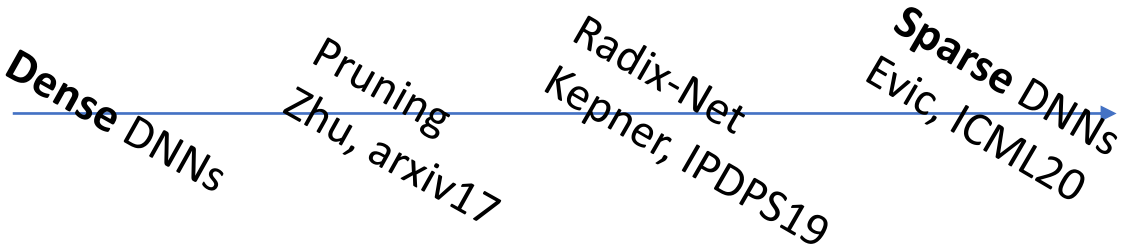


Sources of Sparsity in DNNs

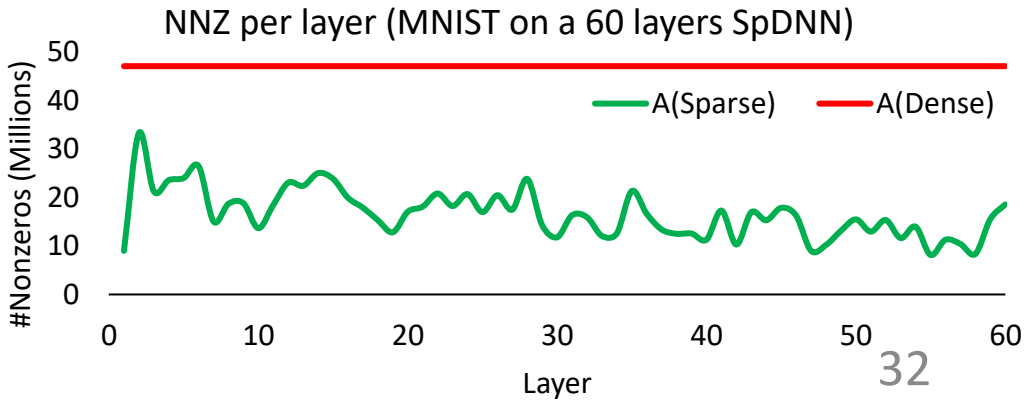
A. Input can be sparse



B. Network can be sparse
- Being pruned during/after training
- Following a predefined sparse architecture
  - TensorFlow Model Optimization

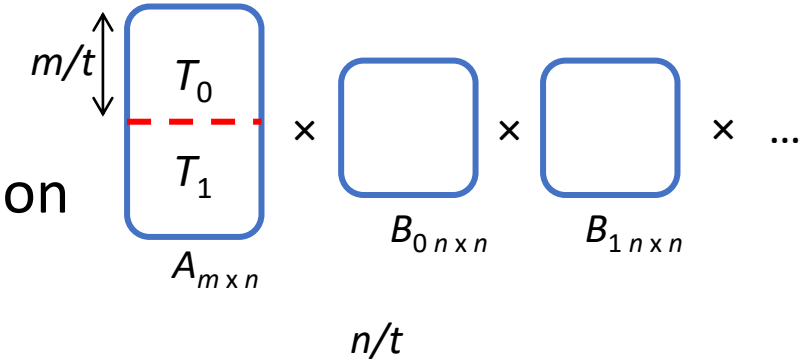


C. Propagating weights can create sparsity
- Due to either input/network sparsity
- Due to the activation function

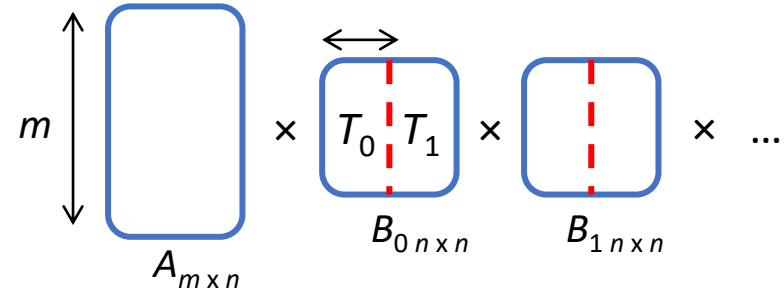


Multithreaded Single Machine (Sparse) DNN Inference

- **Data Parallelism**
  - Horizontal 1D-Row partitioning of input (A)
    - *t* input partitions where *t* is the number of threads
  - \+ No synchronization, – stragglers, – bad L3 utilization



- **Model Parallelism**
  - Vertical 1D-Column partitioning of network (B)
    - *t* network partitions
  - – Strict synchronization, + better L1 & L3 utilization



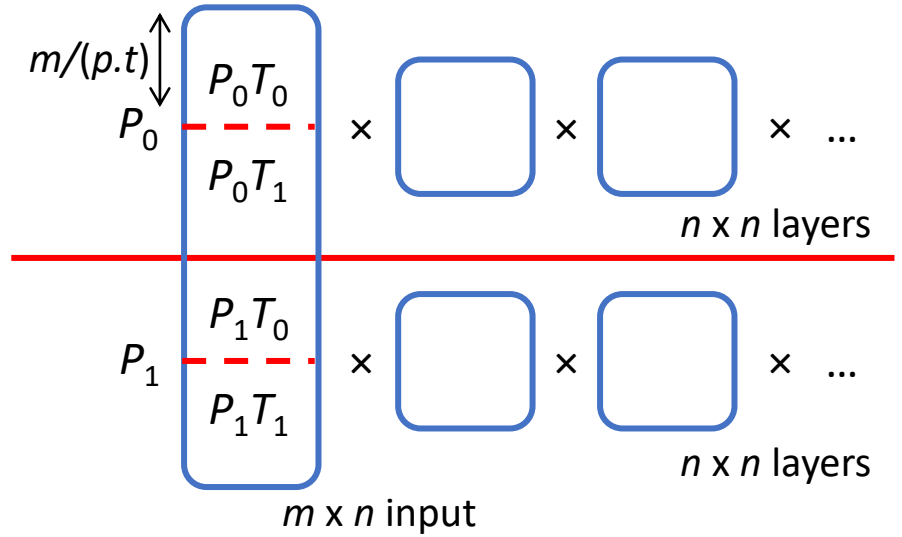
- Each tile is stored using a compressed sparse format

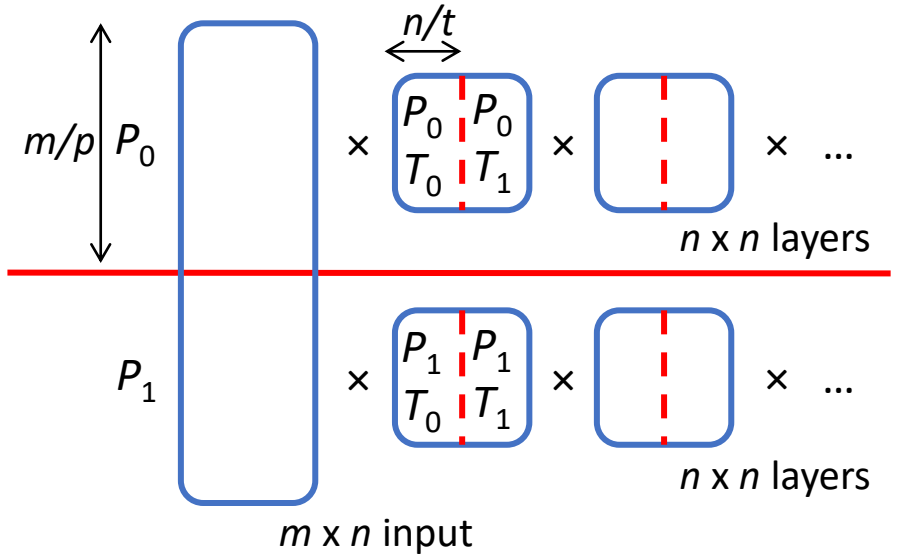
Distributed (Sparse) DNN Inference

- **Data * Data parallelism**
  - Horizontal 1D-Row partitioning of input (A)
  - *p.t* input partitions
  - *p* is the number of processes

- **Data * Model Parallelism**
  - vertical 1D-Column partitioning of network (B)
  - *p.t* network partitions

- Network is replicated for each process
- No communication is happening among processes





Neural Network Hashing to Achieve Load Balance & Locality

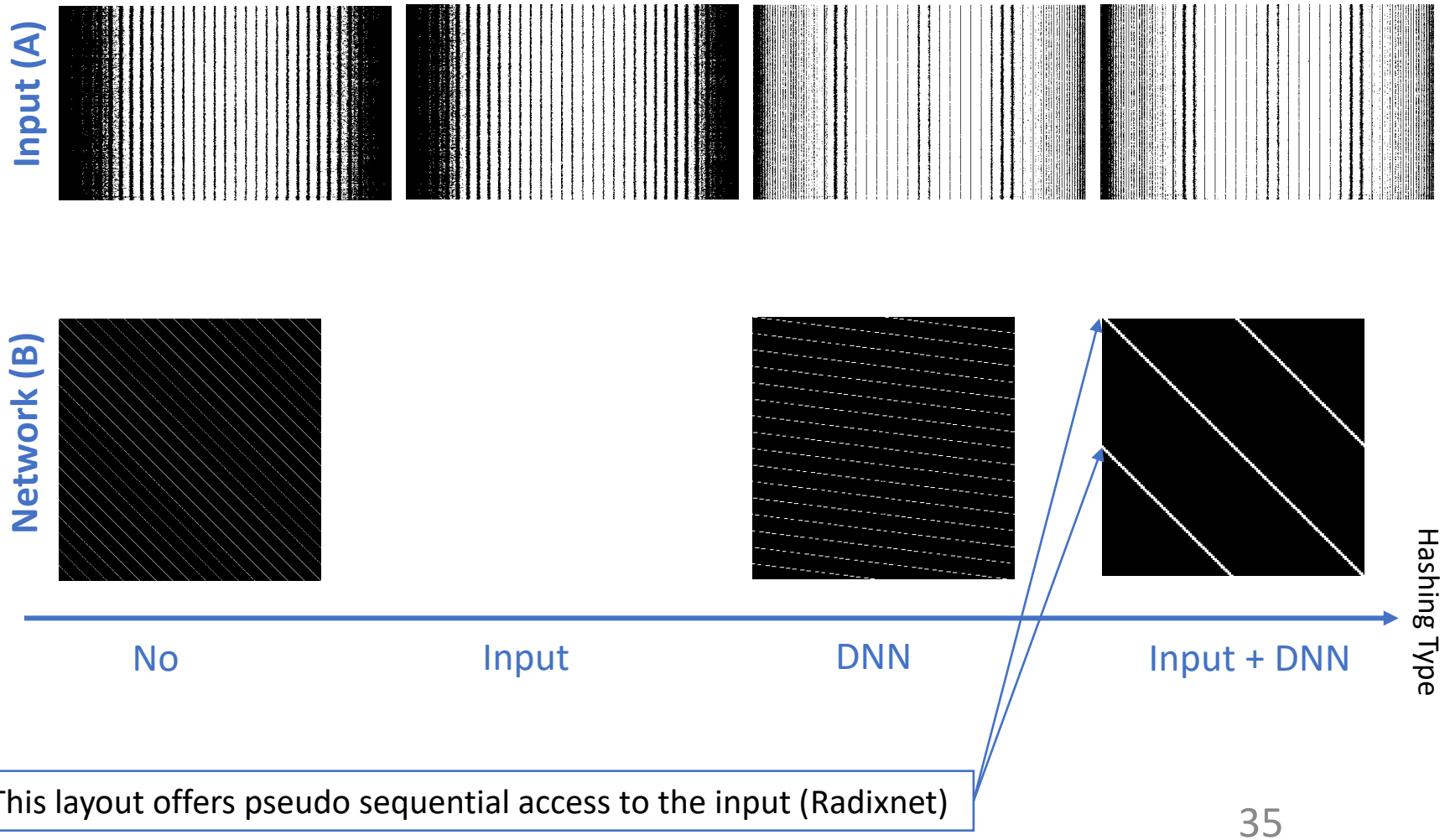
- Hashing mitigates data parallelism stragglers
- **Input hashing**:
  - Hashes rows of A

  + Load balance in data parallelism
- **Network hashing**:
  - Hashes columns of A & rows of B

  + Better access pattern in model parallelism in favor of the cache
- **Input + network hashing**:
  - Hashes rows of A
  - Hashes columns of A & rows of B
  - Hashes columns of B

  + Both above advantages

  + Pseudo sequential access pattern (log(n))



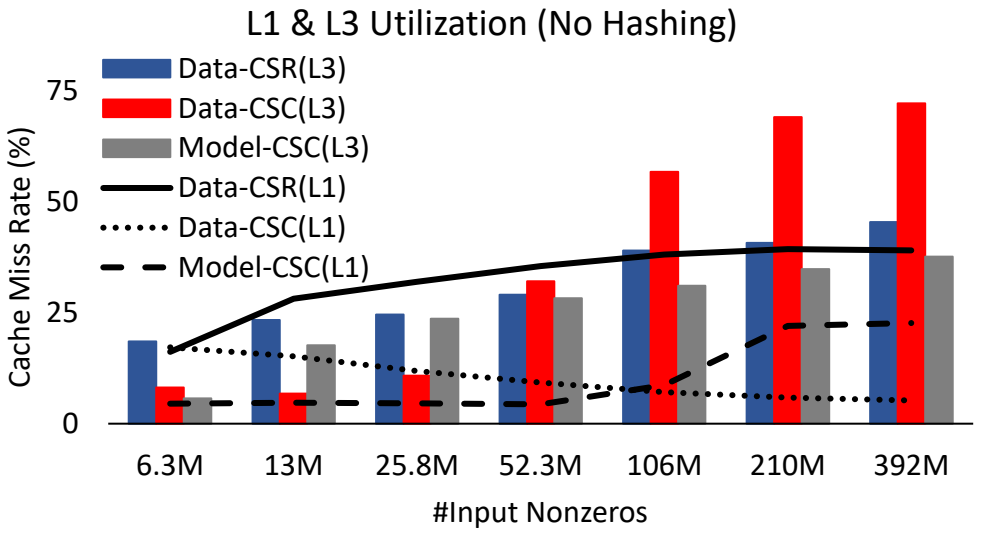
Experiments

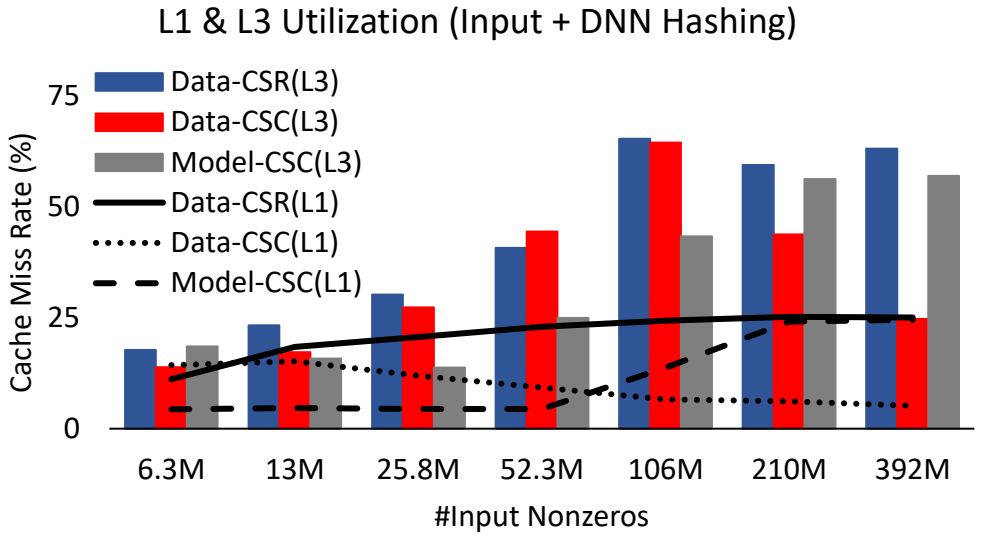
Dataset (RadixNet Sparse DNN, MNIST)						
Input			Network			
#	Size $_{m \times n}$	NNZ	Each Layer		All Layers	
			Size $_{n \times n}$	NNZ	L	NNZ
A_0					120	3.9M
A_1	60K x 1K	6.3M	1K x 1K	32K	480	15.7M
A_2					1920	62.9M
B_0					120	15.7M
B_1	60K x 4K	25M	4K x 4K	131K	480	62.7M
B_2					1920	251M
C_0					120	62.9M
C_1	60K x 16K	99M	16K x 16K	524K	480	251M
C_2					1920	1B
D_0					120	251M
D_1	60K x 65K	392M	65K x 65K	21 M	480	1B
D_2					1920	4B

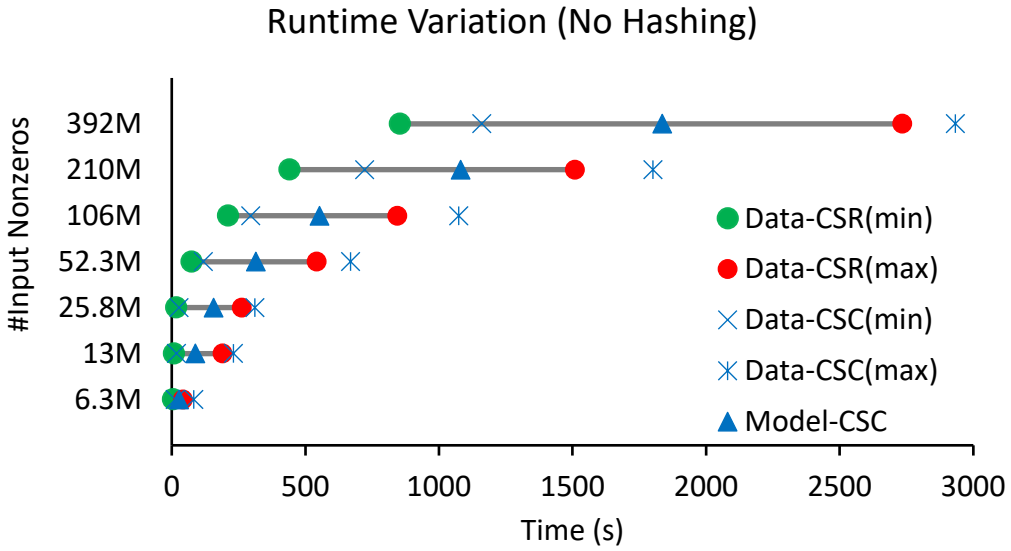
Node Specification (32)	
CPU	28-core @ 2.6 GHZ
Memory	192 GB
OS	Linux
MPI	Intel
Network	Intel Omni-path Fabric

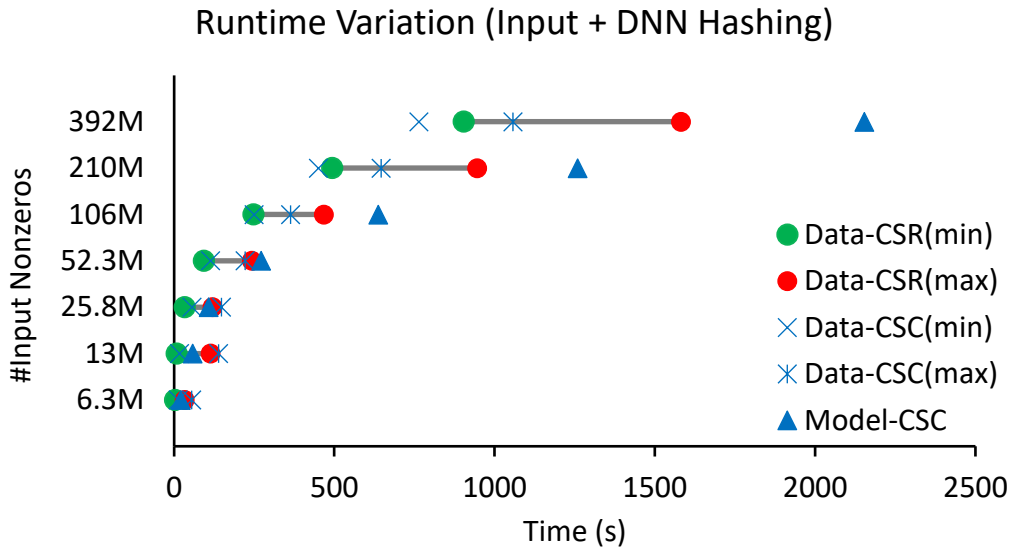
Parallelism/ Multiplication	Right Multiplication (CSC)	Left Multiplication (CSR)
Data Parallelism	Yes	Yes
Model Parallelism	Yes	N/A

Experiments: Comparison of SpMMs (Single)



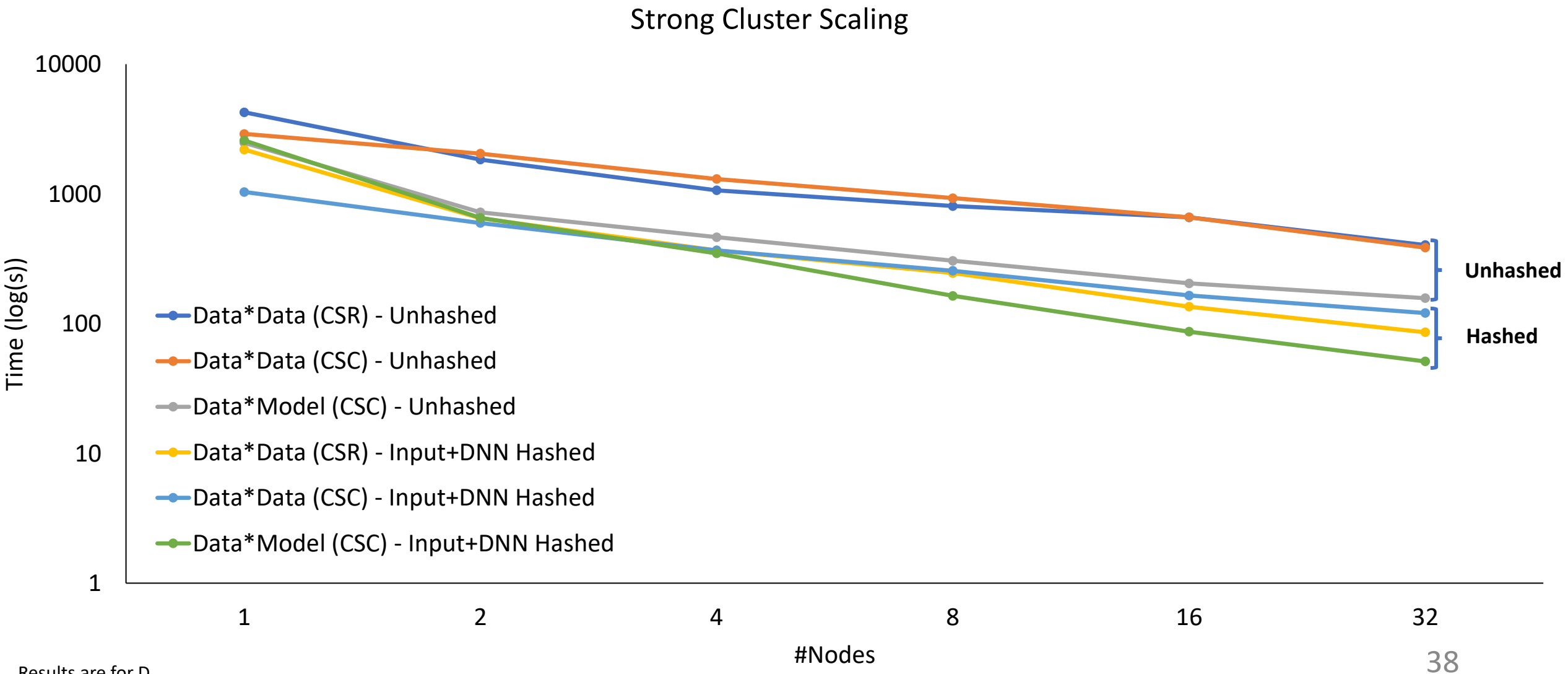






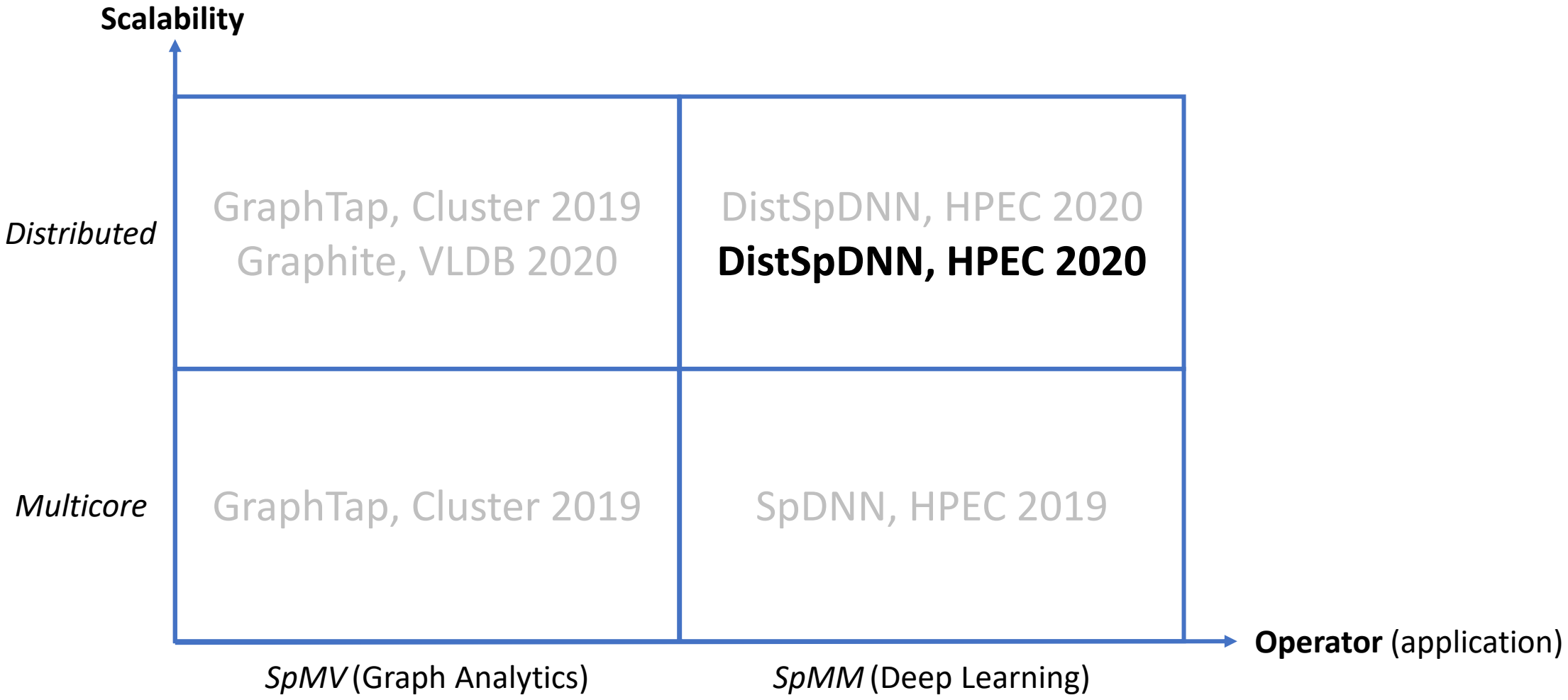
Results are for D_2

Experiments: Comparison of SpMMs (Distributed)



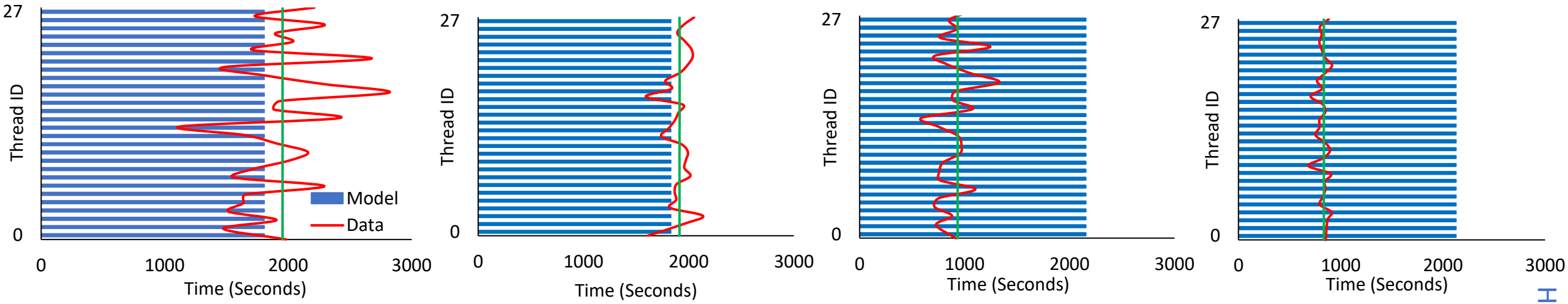
Results are for D_2

Contributions Overview



Motivation: Data Versus Model Parallelism

- Due to imbalance Data parallelism suffers from straggler effect
  - Hashing alleviates the imbalance problem of data parallelism

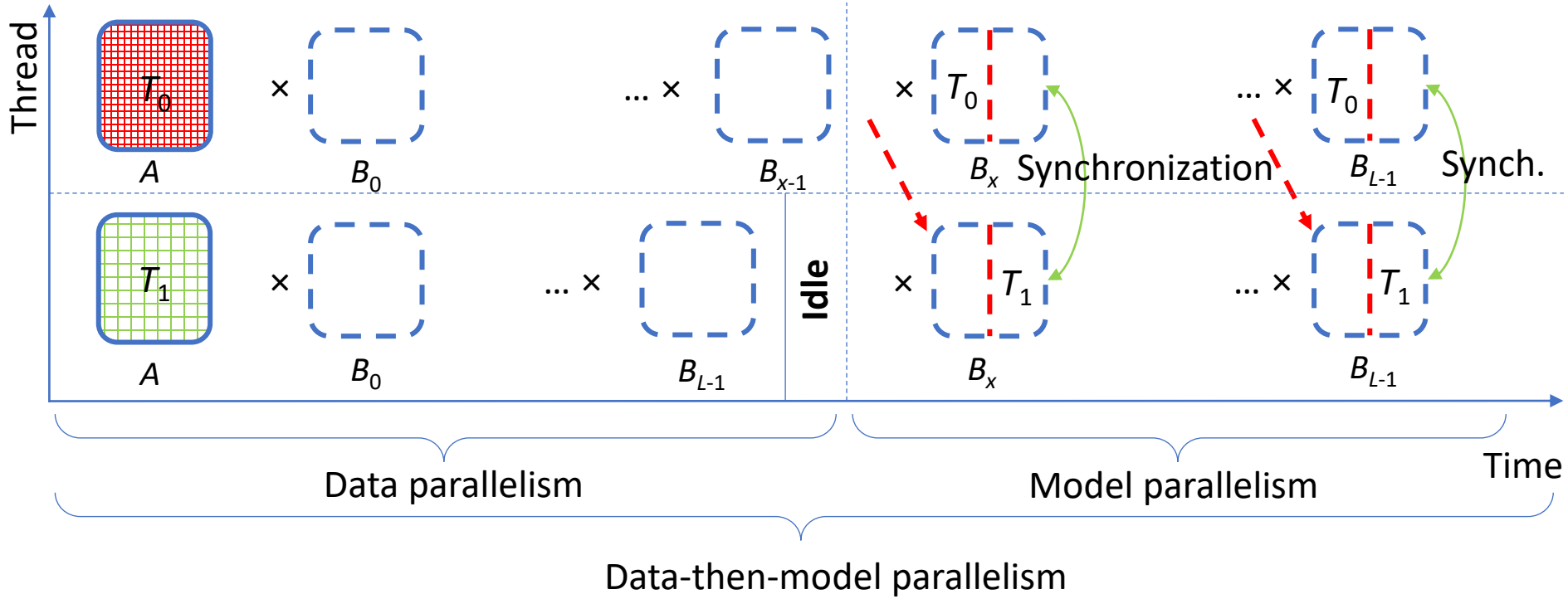


Results are for D_2

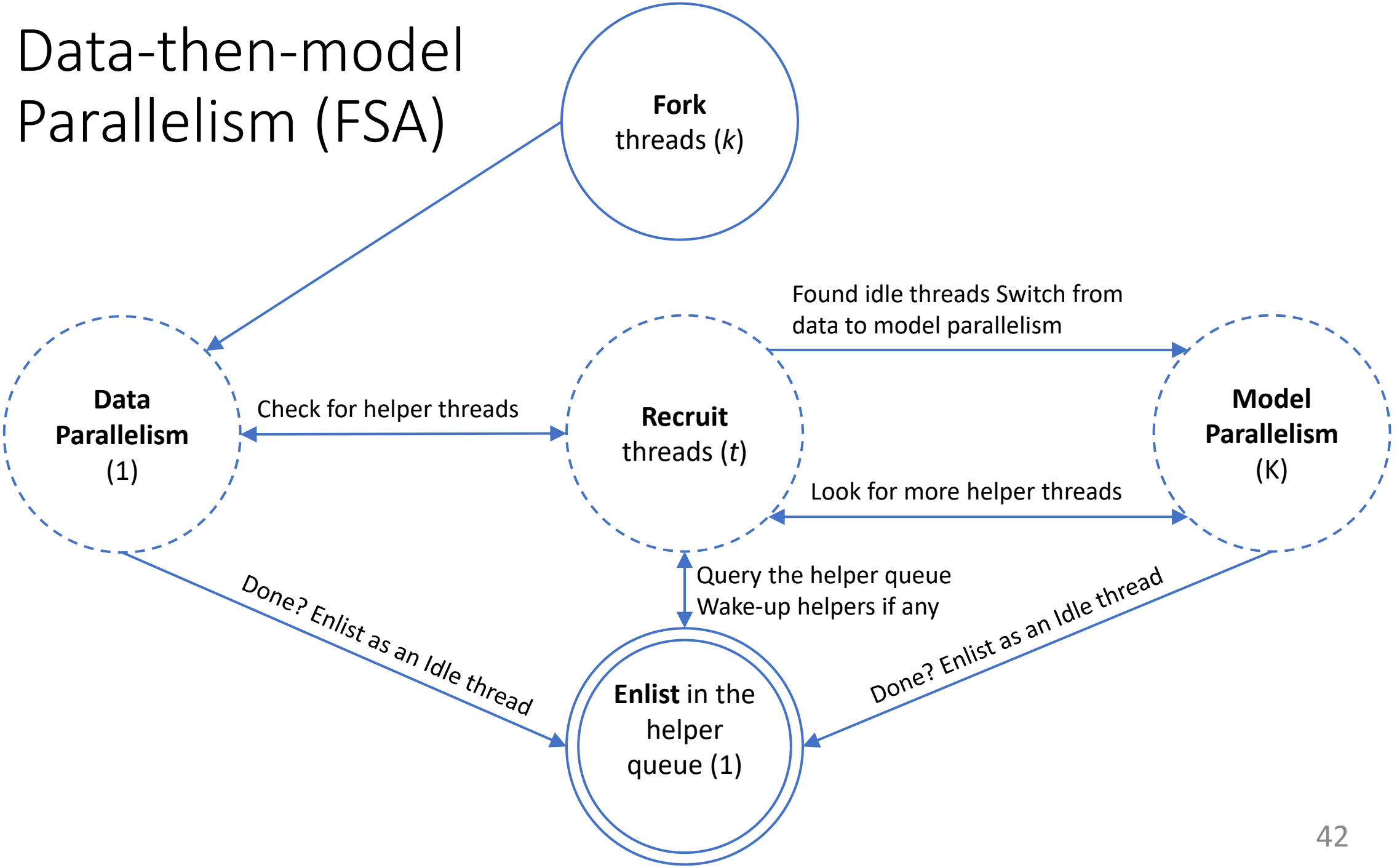
Data-then-model Parallelism

- Switch from data to model parallelism and turning idle threads into additional processing power to mitigate the effect of stragglers

- Lazy load balancing by reusing idle threads
- Less synchronization cost



Data-then-model Parallelism (FSA)



Experiments

Dataset (RadixNet Sparse DNN, MNIST)						
Input			DNN			
			Each Layer		All Layers	
#	Size $_{m \times n}$	NNZ	Size $_{n \times n}$	NNZ	L	NNZ
A_0					120	3.9M
A_1	60K x 1K	6.3M	1K x 1K	32K	480	15.7M
A_2					1920	62.9M
B_0					120	15.7M
B_1	60K x 4K	25M	4K x 4K	131K	480	62.7M
B_2					1920	251M
C_0					120	62.9M
C_1	60K x 16K	99M	16K x 16K	524K	480	251M
C_2					1920	1B
D_0					120	251M
D_1	60K x 65K	392M	65K x 65K	21 M	480	1B
D_2					1920	4B

Node Specification (16)	
CPU	28-core @ 2.6 GHZ
Memory	192 GB
OS	Linux
MPI	Intel
Network	Intel Omni-path Fabric

Parallelisms

Data Parallelism

Model Parallelism

Data-then-model Parallelism

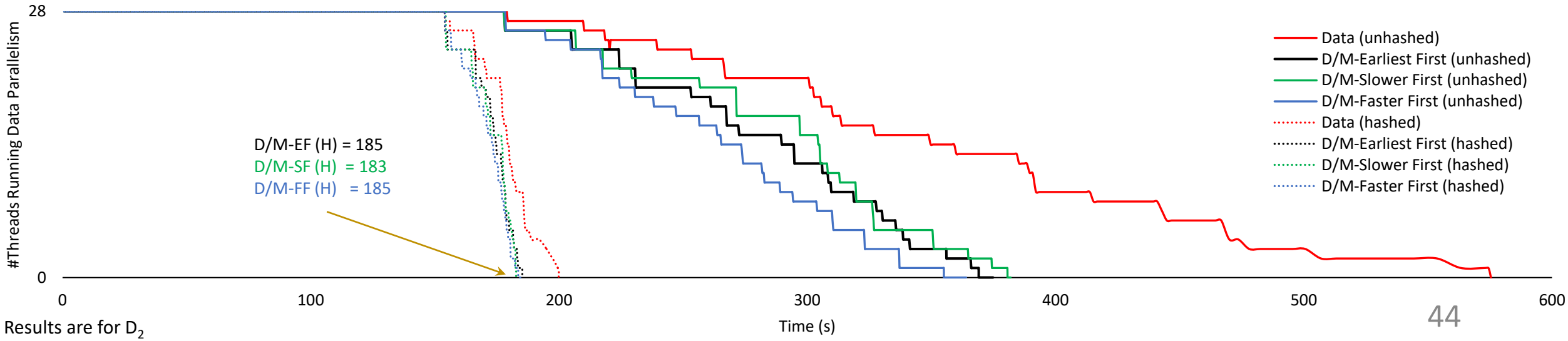
Manager-worker Parallelism

Work-stealing Parallelism

Experiments: Data-then-model Parallelism Thread Scheduling Algorithms

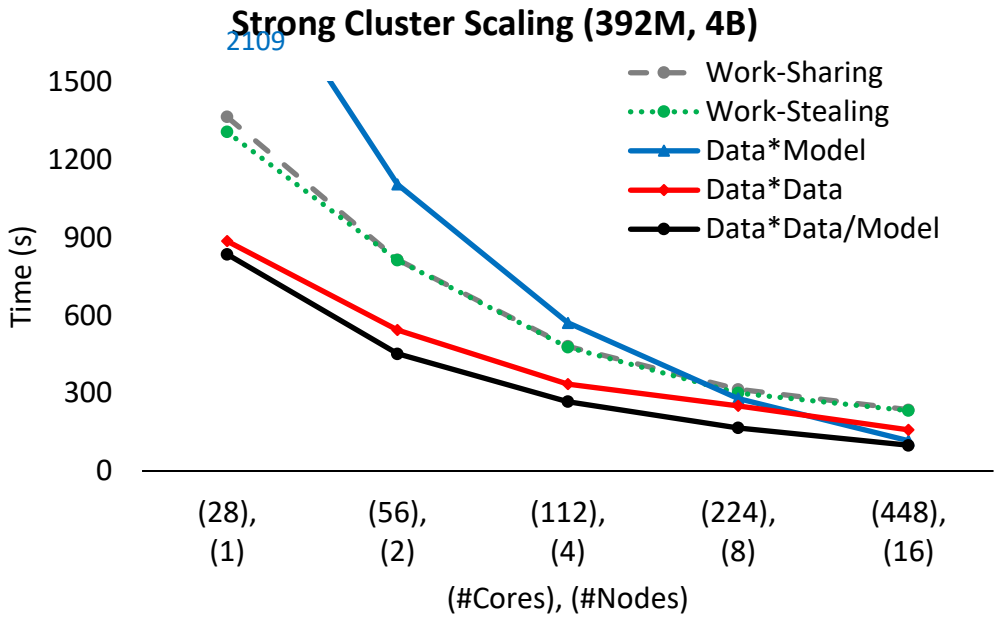
- **Locking Mechanism Runtime**
  - Threads are either
    - Worker (active) or helper (newly idle)
  - An idle thread enlists into an **idle queue**
  - A **working thread** probes the idle queue
    - **Helper threads** get recruited by a working one

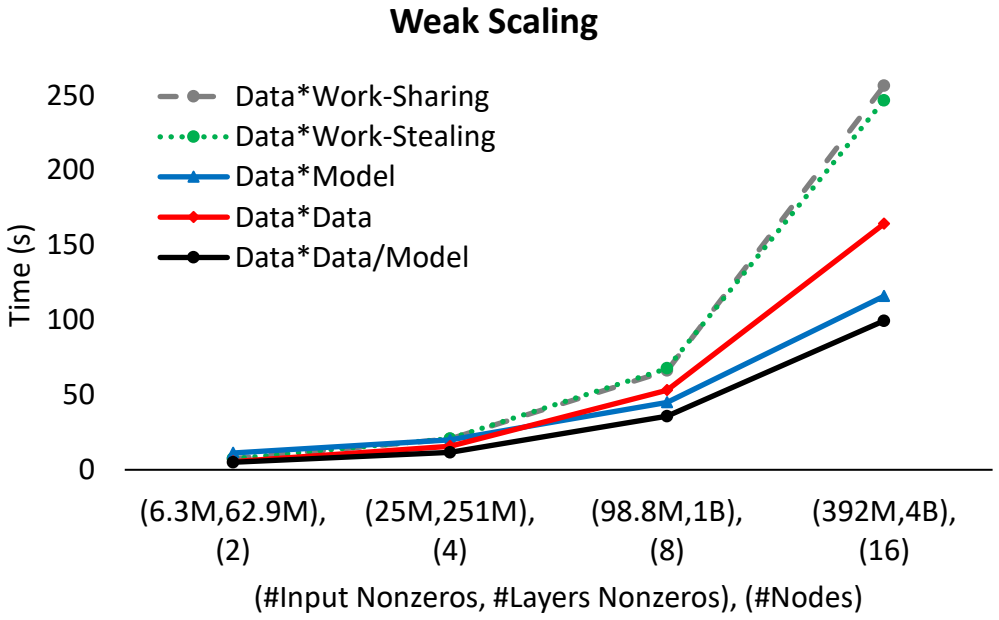
- **Advantages**
  - + Overloadable with scheduling strategies
    - Earliest first, slower first, and faster first
  - + Fully decentralized and asynchronous
  - + Minimal lock contention
    - condition variables + locks
  - + Elastic: adding/removing threads on the fly
    - zero data movement

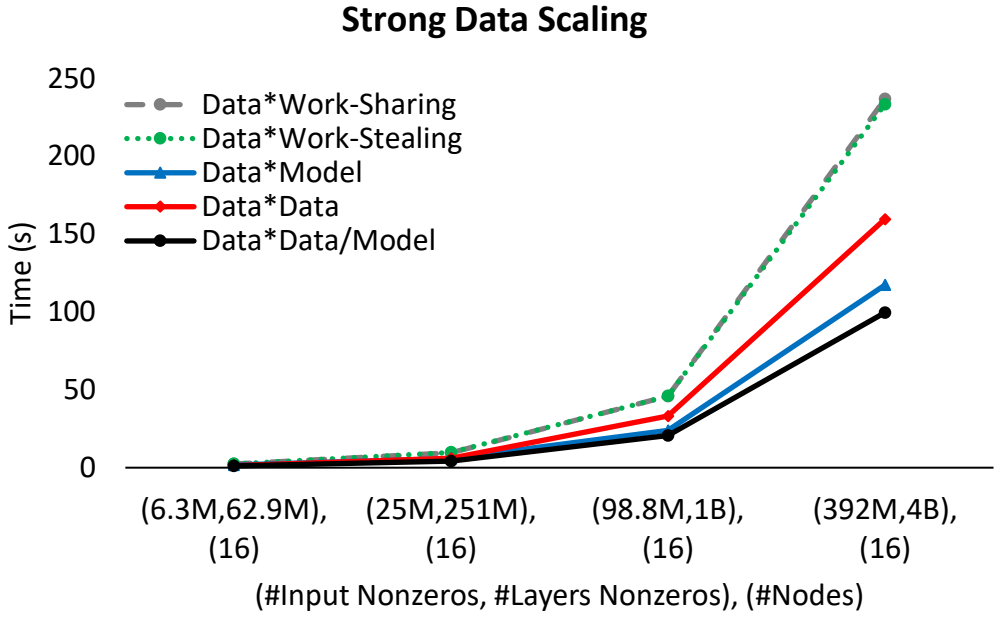


Results are for D_2

Scalability Experiment







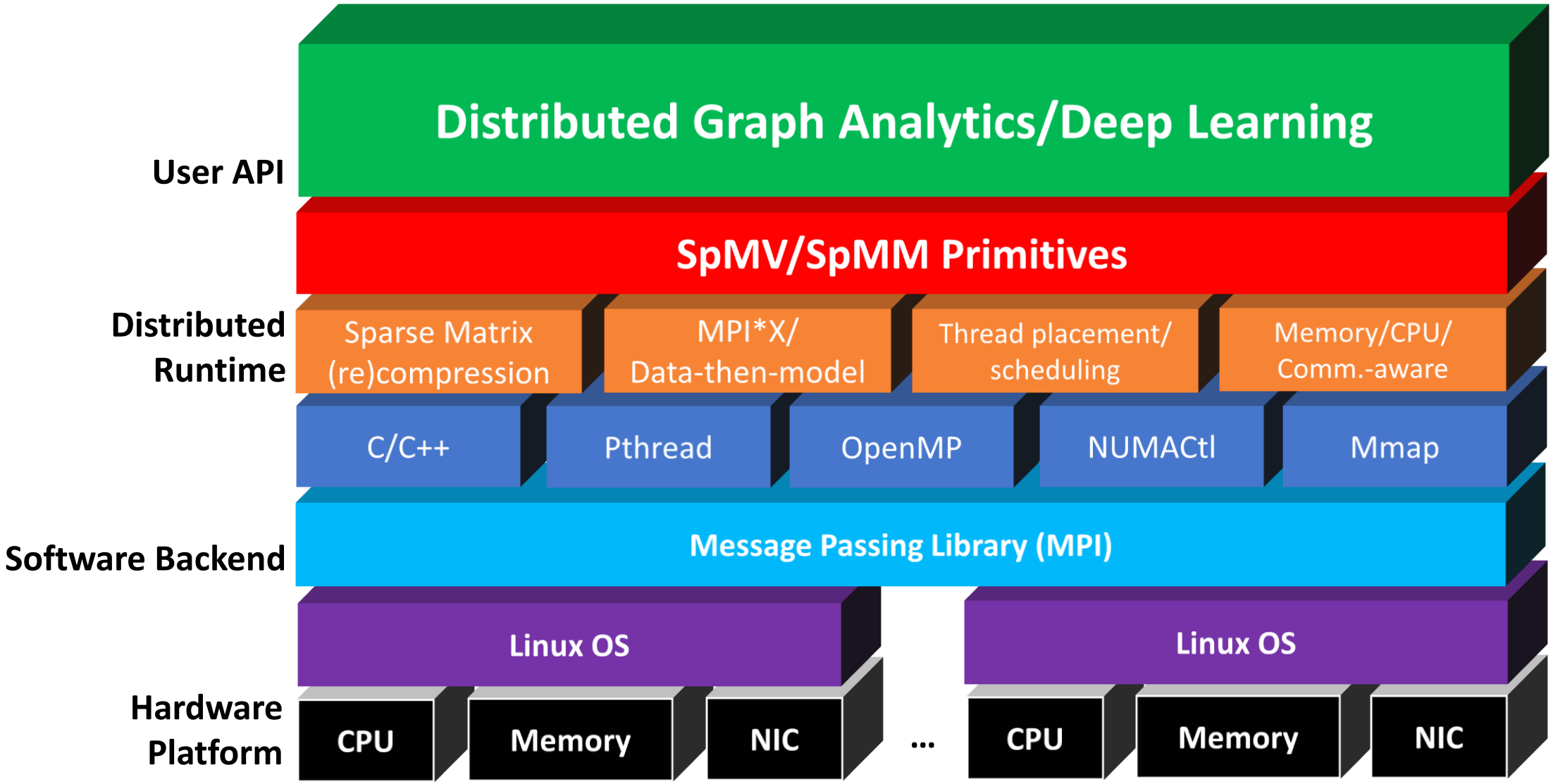
SpDNN & DistSpDNN vs. State-of-the-art

System	Computing Model	Platform	Parallelism Model	Computation & Communication	Matrix Compression	Authors
TensorFlow	Linear Algebra	Commodity/ HPC/ Cloud	Data parallelism (Keras + CPU/GPU/TPU)	Horovod	Dense	2015 – present, Google
PyTorch	Linear Algebra	Commodity/ HPC	Data parallelism (MPI/ Gloo + OpenMP/ CUDA)	Horovod	Dense	2016 – present, Facebook
MXNet	Linear Algebra	Commodity/ HPC	Data parallelism (MPI4Py/ Gloo + OpenMP/ CUDA)	Horovod/ TCP/IP socket	Dense	2020 – present, Amazon
DeepSpeed	Linear Algebra	Commodity/ HPC/ Cloud	Data parallelism (MPI/ NCCL + X/ CUDA)	Communication overlapping	Sparse	2019 – present, Microsoft
SpDNN	Linear Algebra	Multicore	Model (OpenMP)		**Sparse**	2019, UPitt
DistSpDNN	Linear Algebra	HPC	**Data-then-model (MPI*OpenMP)**	**NUMA-aware Memory re(allocation)**	**Sparse, 2-step SpMM**	2020, UPitt

Summary & Conclusion (1)

- Sparse Matrix – Vector (SpMV)
  - **TCSC** that reduces time, space, and communication complexities (GraphTap, Cluster 2019)
    - Co-compression reduces the communication volume
  - **MPI*X** that deems threads as basic unit of computation, computation, and synchronization (Graphite, VLDB 2020)
    - Better scalability requires laying out the computation/communication path beforehand
- Sparse Matrix – Matrix Multiplication (SpMM)
  - **Neural network hashing** for alleviating the load imbalance and offering cache locality (DistSpDNN, HPEC 2020)
    - Proper hashing can offer super-linear speedup (log(*n*) time)
  - **Data-then-model parallelism** for mitigating the straggler effect (DistSpDNN, HPEC 2020)
    - Sparse computations require new parallelism that takes account of the runtime imbalance

Summary & Conclusion (2)



Implementations are available at https://github.com/hmofrad

Acknowledgment!

Questions?