BERKELEY LAB
Bringing Science Solutions to the World

U.S. DEPARTMENT OF ENERGY
Office of Science

Distributed-Memory Sparse Kernels for Machine Learning

Vivek Bharadwaj*, Aydın Buluç*†, James Demmel*

*EECS Department, UC Berkeley

†Computational Research Division, Lawrence Berkeley National Laboratory

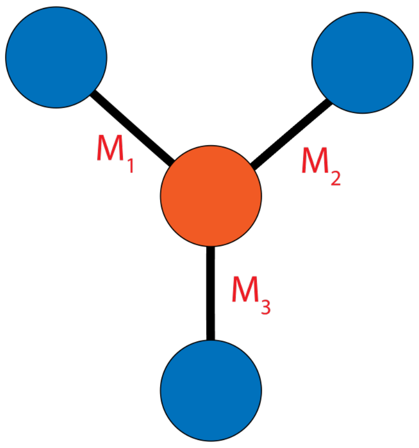
IPDPS

Sparse Kernels in Machine Learning

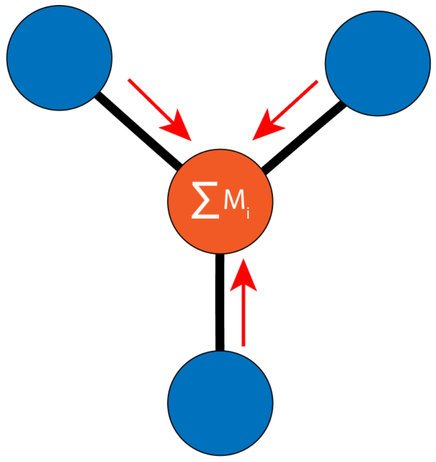
- Sampled Dense-Dense Matrix Multiplication (SDDMM) and Sparse-times-Dense Matrix Multiplication (SpMM) appear in a variety of applications:
  - Graph Neural Networks with Self-Attention
  - Collaborative Filtering with Alternating Least Squares
  - Document Clustering by Wordmover's Distance

- Both kernels involve a single sparse matrix and two (typically tall-skinny) dense matrices. Typically, applications use both operations in sequence.

- When the sparse matrix is the adjacency matrix of a graph, we interpret the kernels as follows:
  - SDDMM generates a message on each edge
  - SpMM aggregates messages from edges incident to each vertex



Message Generation



Message Aggregation

Existing Work

Shared Memory SDDMM, SpMM, FusedMM

Cache-aware tiling: Sampled Dense Matrix Multiplication for High-Performance Machine Learning: Nisa et al. (HiPC 2018)

Sparse Matrix Reordering: Adaptive sparse tiling for sparse matrix multiplication: C. Hong et al. (PPOP 2019)

Tile Shape Tuned to Sparsity: A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs: P. Jiang et al. (PPOP 2020)

Local SDDMM / SpMM Kernel Fusion: FusedMM: A Unified SDDMM-SpMM Kernel for Graph Embedding and Graph Neural Networks: M. K. Rahman et al. (IPDPS 2021)

Distributed Memory Dense GEMM

Optimize for Extra Memory: Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms: E. Solomonik and J. Demmel
(EuroPar 2011)

Optimized Schedules for Non-Square GEMM: Red-blue pebbling revisited: Near optimal parallel matrix-matrix multiplication: G. Kwasniewski (SC 19)

Distributed Sparsity-Agnostic SpMM

1.5D Algorithms on Square Matrices: Communication-Avoiding Parallel Sparse-Dense Matrix-Matrix Multiplication: P. Koanantakool et al. (IPDPS 2016)

1.5D Algorithms embedded in GNNs: Reducing communication in graph neural network training: Tripathy et al. (SC 20)

1.5D and 2D Algorithms, One-Sided Communication: Distributed-memory parallel algorithms for sparse times tall-skinny-dense matrix multiplication: Selvitopi et al. (ICS 21)

Distributed SDDMM and FusedMM

?

Our Contributions

- We design the **first distributed-memory implementations of SDDMM** based on communication-avoiding algorithms for SpMM in the literature. Our implementations benefit from additional memory by replicating inputs and outputs.
- We give **strategies to elide communication when executing SDDMM and SpMM in sequence** (FusedMM), eliminating communication and changing the optimal replication factor for both kernels.
- We benchmark our algorithms on hundreds of nodes of LBNL Cori, testing with both Erdos-Renyi random matrices and billion-scale real-world matrices.

Distributed-Memory SDDMM Algorithms

Symbols

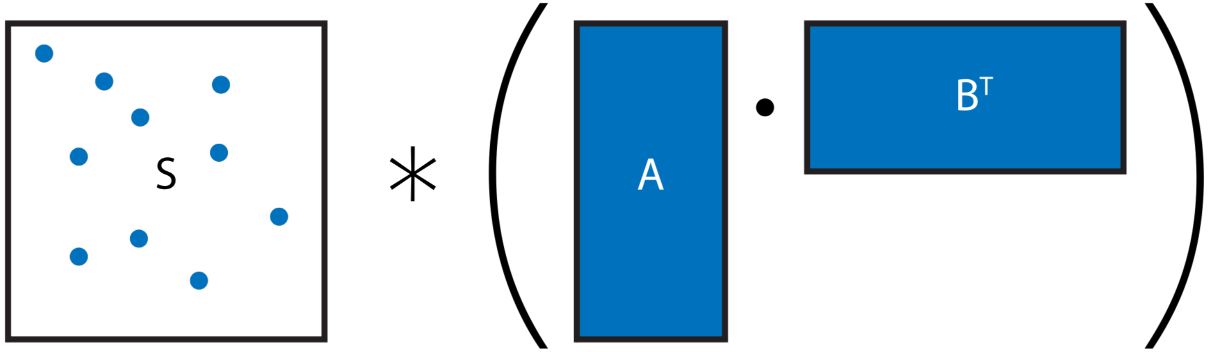
Symbol	Definition
S, R	$m \times n$ sparse matrices
A	$m \times r$ dense matrix
B	$n \times r$ dense matrix
ϕ	The ratio $\mathrm{nnz}(S)/nr$
$*$	Elementwise multiplication
$\cdot$	Matrix Multiplication

Symbols and Definitions

- Given dense matrices A, B of dimensions $m \times r$, $n \times r$, respectively, and a sparse matrix S of dimensions $m \times n$, define **Sampled Dense-Dense Matrix Multiplication** as:

$$\mathrm{SDDMM}(S, A, B) \coloneqq \mathrm{S} * (\mathrm{A} \cdot B^T)$$

- Output has nonzero locations identical to S

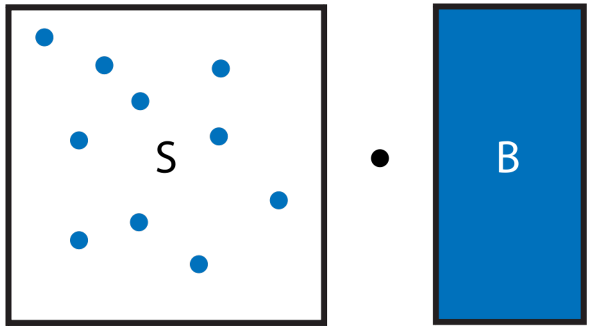


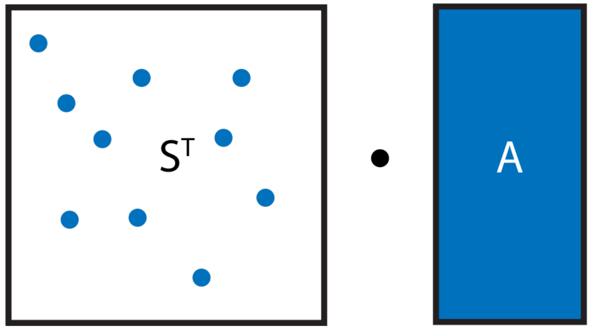
Symbols and Definitions

- We distinguish between the SpMM operation that multiplies S and A and the operation that multiplies S^T and B. GNNs, collaborative filtering require **both**.

- Define **SpMMA, SpMMB** as:

$$\mathrm{SpMMA}(S, B) \coloneqq S \cdot B \qquad \mathrm{SpMMB}(S, A) \coloneqq S^T \cdot A$$





Symbols and Definitions

- Applications typically make a call to SDDMM (message generation) and feed the sparse output directly to an SpMM operation (message aggregation)

- Define **FusedMMA, FusedMMB** as compositions of SDDMM with SpMMA, SpMMB

$$\mathrm{FusedMMA}(S, A, B) \coloneqq \mathrm{SpMMA}(\mathrm{SDDMM}(\mathrm{S}, \mathrm{A}, \mathrm{B}), \mathrm{B})$$

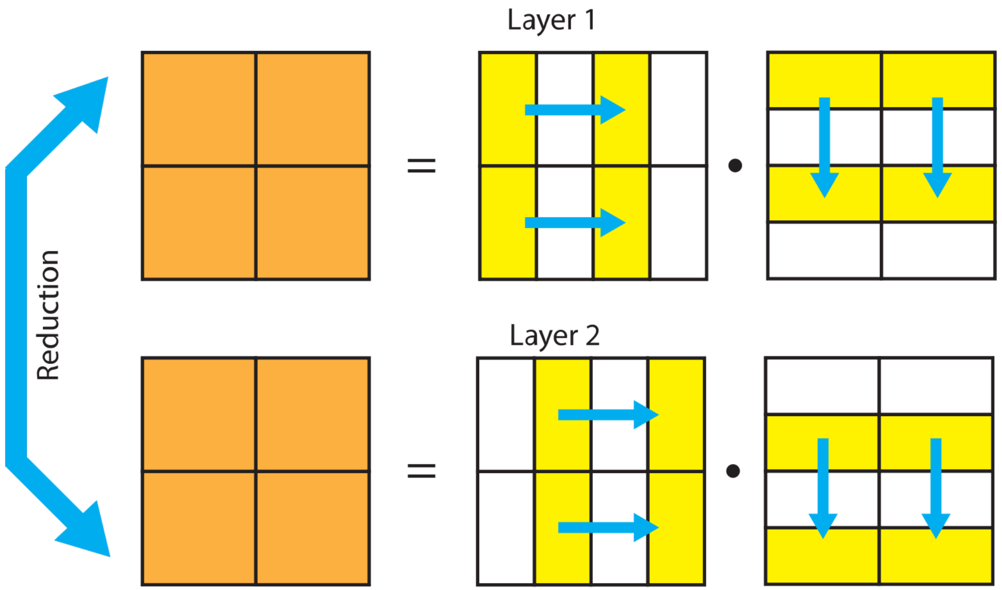
$$\mathrm{FusedMM}B(S, A, B) \coloneqq \mathrm{SpMMB}(\mathrm{SDDMM}(\mathrm{S}, \mathrm{A}, \mathrm{B}), \mathrm{A})$$

Sparsity-Agnostic Distributed SpMM

- Sparsity-agnostic algorithms operate similarly to distributed dense GEMM algorithms (Cannon, SUMMA) by shifting large blocks A, B, and S.

- Do not benefit from graph partitioning, rely on random permutations of the rows and columns of S.

- We categorize such algorithms by the choice of which submatrices they **replicate, propagate,** and keep **stationary**



2.5D Algorithm for Dense GEMM
8 Processors, Replication Factor 2

Converting SpMM Algorithms to SDDMM Algorithms

- SDDMM and SpMM have **identical data access patterns**. Consider serial algorithms for both kernels:

$\mathrm{R} \coloneqq \mathrm{SDDMM}(S, A, B)$

for $(i, j) \in S$
$$R_{ij} \coloneqq S_{ij}(A_{i:} \cdot B_{j:}^{T})$$

$\mathrm{A} \coloneqq \mathrm{SpMMA}(S, B)$

for $(i, j) \in S$
$$A_{i:} \mathrel{+}= S_{ij} B_{j:}$$

- Every nonzero (i, j) requires an interaction between row i of A and row j of B. As a result:

Every distributed algorithm for SpMM can be converted to an algorithm for SDDMM with identical communication characteristics, and vice-versa.

Converting SpMM Algorithms to SDDMM Algorithms

- Consider any distributed algorithm for SpMMA that performs no replication. For all indices $k \in [1, r]$, the algorithm must (at some point)
  - Co-locate S_{ij}, A_{ik}, B_{jk} on a single processor
  - Perform the update $A_{ik} \mathrel{+}= S_{ij} B_{jk}$

- Transform this algorithm as follows:

1. Change the input sparse matrix S to an output that is initialized to 0.
2. Change A from an input to an output.
3. Have each processor execute the local update: $S_{ij} \mathrel{+}= A_{ik} B_{jk}$

The resulting algorithm performs SDDMM (up to multiplication with the values initially in S) with communication characteristics and data layout identical to the original.

Converting SpMM Algorithms to SDDMM Algorithms

- 1.5D and 2.5D SpMM algorithms replicate input / output matrices to reduce communication bandwidth (using extra memory)

- Inputs typically replicated via broadcast at the beginning of the algorithm

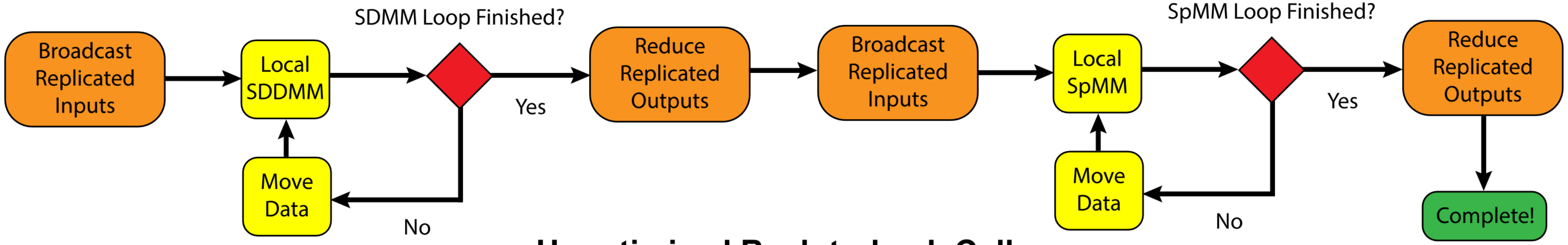
- **Reduction** required at the end of the algorithm to sum up temporary accumulation buffers

- We extend our transformation procedure to algorithms with replication by:
  - Replacing initial broadcasts of input buffers with terminal reductions of those buffers
  - Replacing terminal reductions of output buffers with initial broadcasts

Communication-Eliding Strategies for FusedMM

A Simple Strategy for Distributed FusedMM

- Consider the FusedMMA operation. The simplest distributed implementation executes the SDDMM and feeds the intermediate result to SpMM
- Identical input / output data layouts let us avoid reorganizing A, B, and S
- Still performs replication, propagation for both SDDMM and SpMM

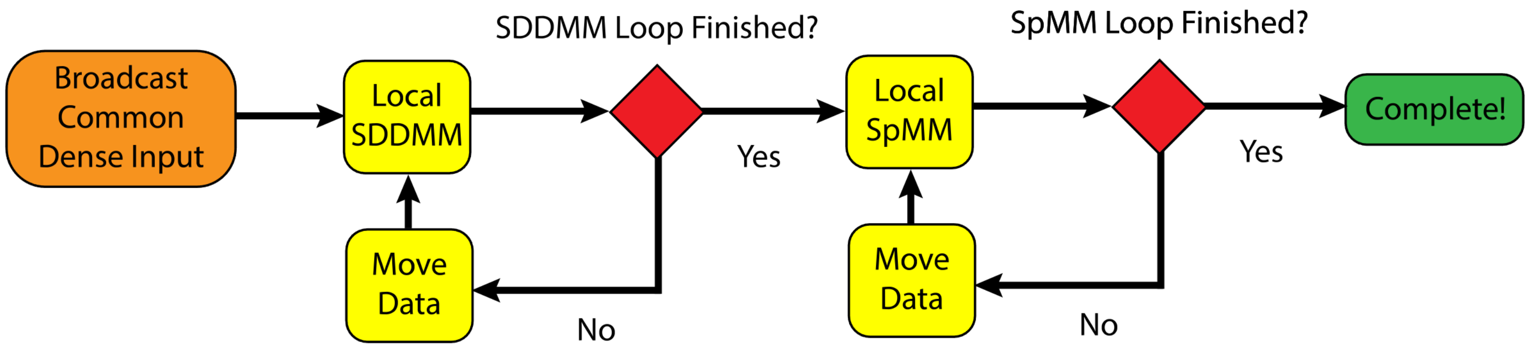


Unoptimized Back-to-back Calls

Communication Elision: Replication Reuse

- We could replicate the same dense input matrix for both SDDMM and SpMM. We call this strategy **replication reuse**

- We save communication by **increasing** the replication factor relative to the unoptimized sequence

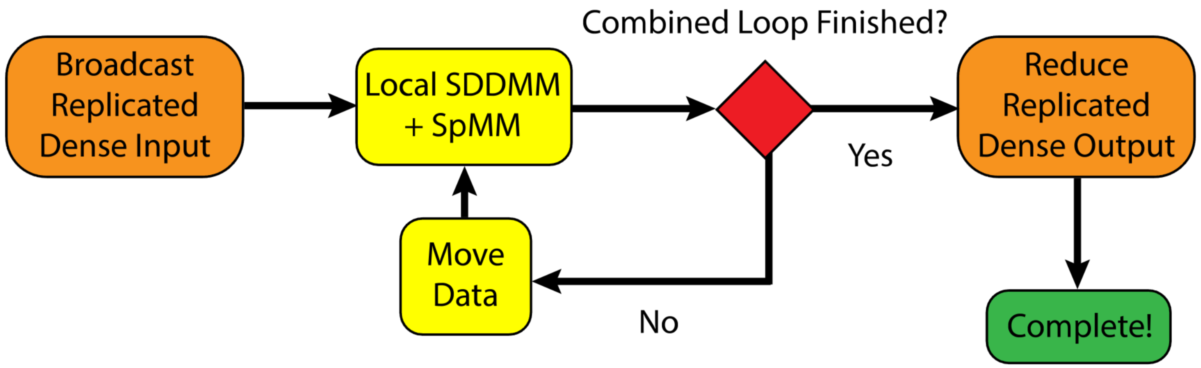


Replication Reuse

Communication Elision: Local Kernel Fusion

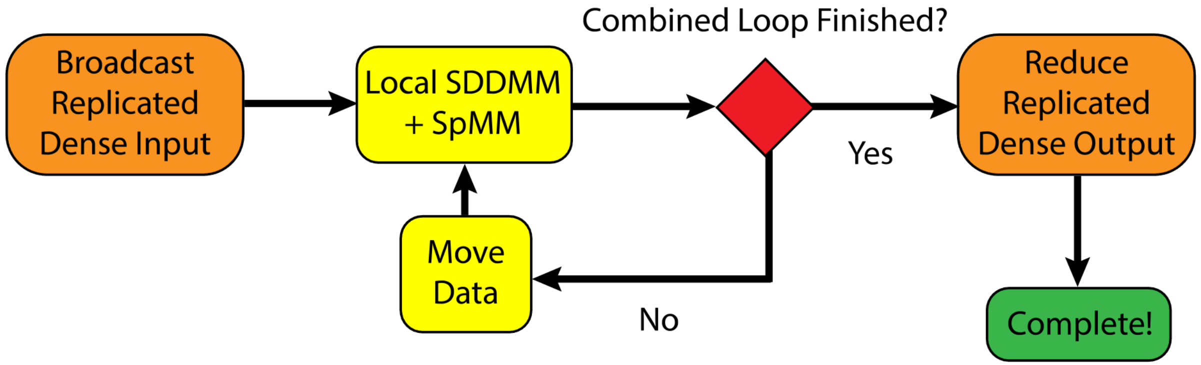
- We could execute a local SDDMM and SpMM on each processor without any intermediate communication. We call this strategy **local kernel fusion**.

- We save communication by **decreasing** the replication factor compared to the unoptimized case



Local Kernel Fusion

Communication Elision: Local Kernel Fusion



Local Kernel Fusion

- **Caveat:** Cannot apply this strategy for any algorithm that splits the dense matrices by columns among processors
- Message generation on each edge **must** precede aggregation. Cannot begin SpMM with partial results on the edges.

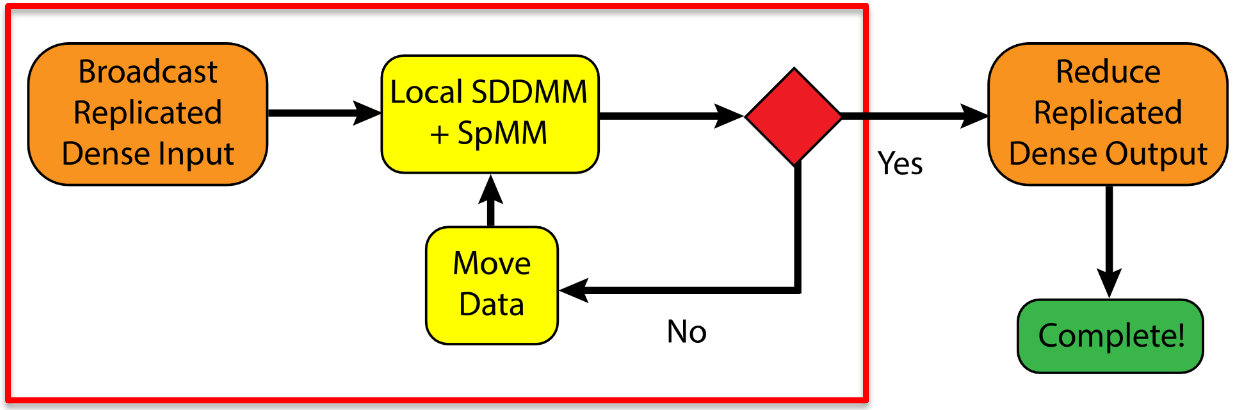
Algorithm Data Movement

Replication and Propagation Choices

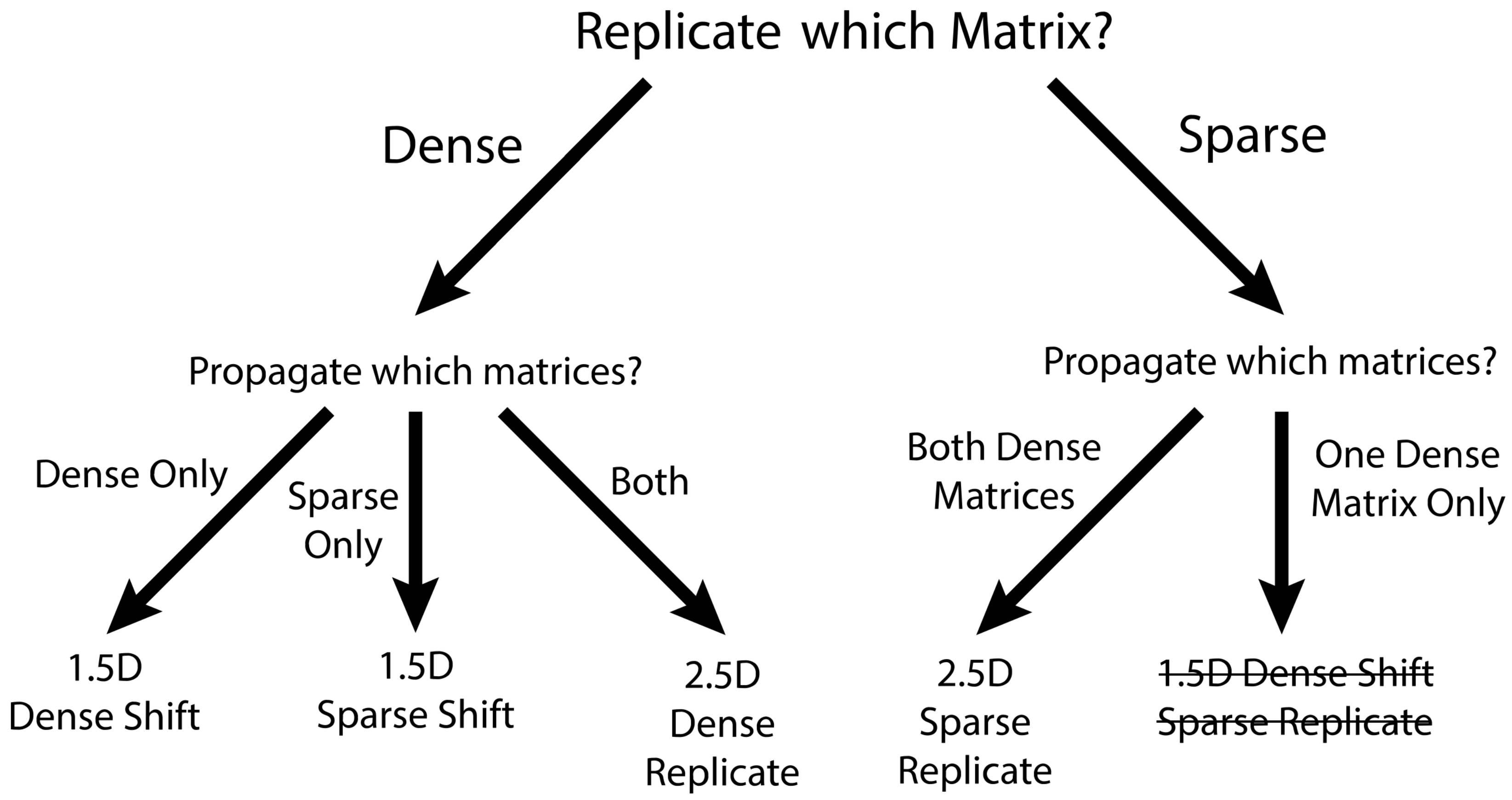
- We design our algorithms by deciding which matrices to **replicate**, **propagate**, and **keep stationary**. For the sake of our communication analysis, assume $m \approx n$.

- These choices affect the communication complexity of each algorithm

- The optimal algorithm choice depends on the ratio between the **nonzero count of the sparse matrix** and the **total entries in either dense matrix,** which we define as ϕ.



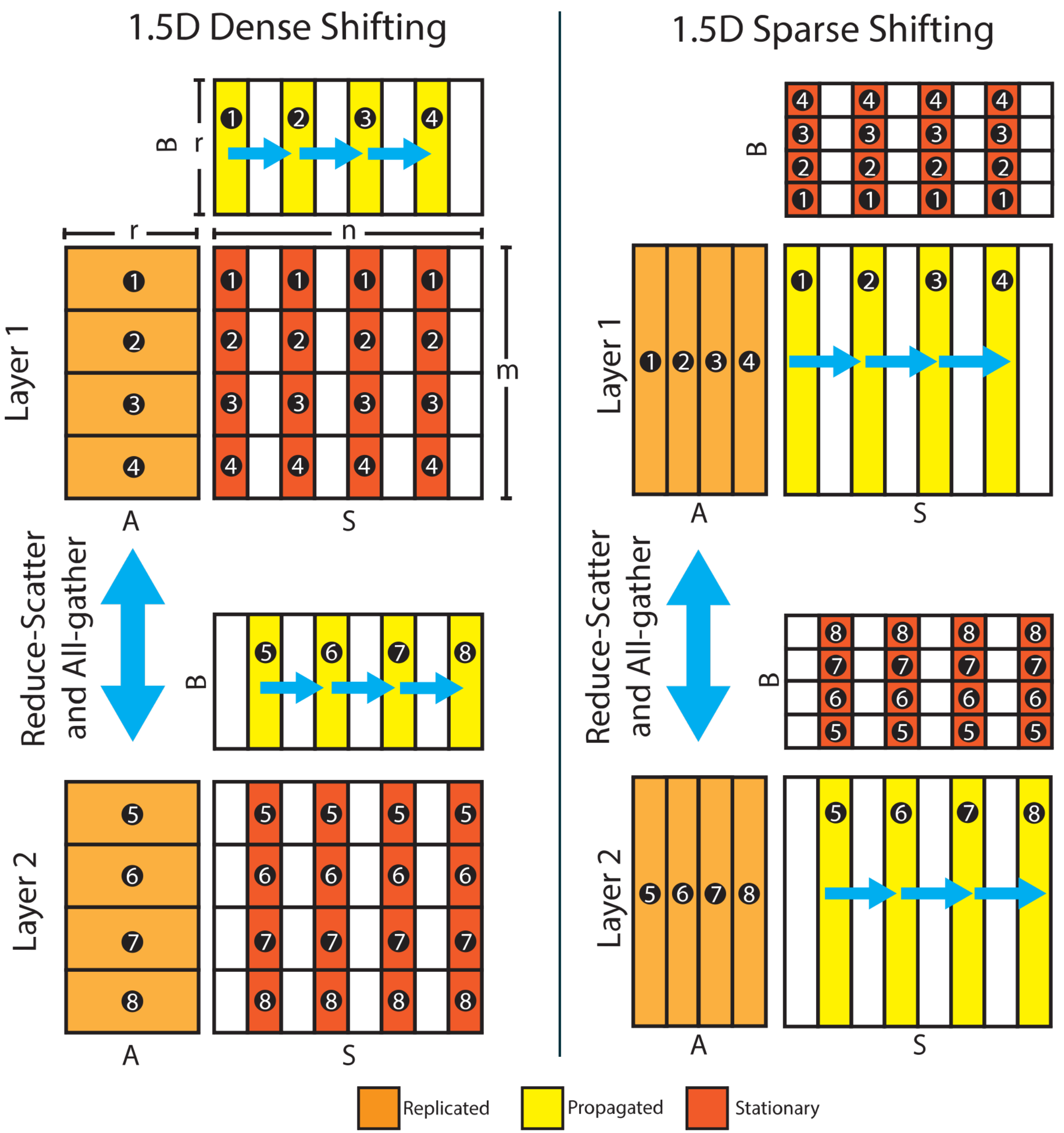
Replication and Propagation Choices



1.5D Algorithms

- Two variants, both replicating a dense matrix:
  - **Cyclically shift the dense matrix**, keep the sparse matrix stationary
  - **Cyclically shift the sparse matrix**, keep the dense matrix stationary
- Choice affects the # of words communicated:

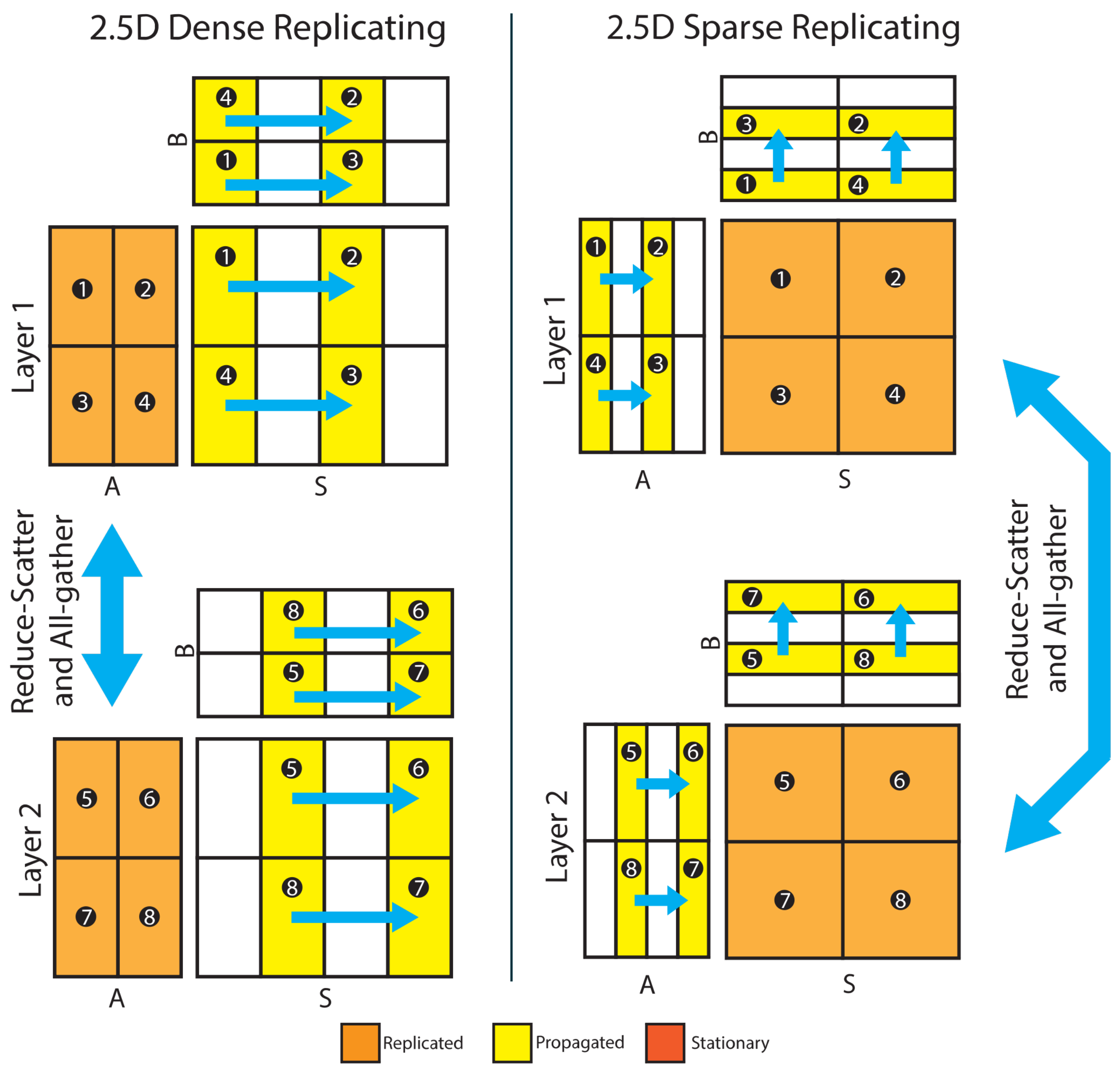
Dense Shift	Sparse Shift
$O\left(\frac{nr}{p^{1/2}}\right)$	$O\left(\frac{nr\phi^{1/2}}{p^{1/2}}\right)$



2.5D Algorithms

- Two variants, both shifting at least one dense matrix:
  - **Replicate one dense matrix,** cyclically shift the other dense matrix and a sparse matrix
  - **Replicate the sparse matrix,** cyclically shift both dense matrices
- # of words communicated:

Dense Replicate	Sparse Replicate
$O\left(\frac{nr\phi^{2/3}}{p^{2/3}}\right)$	$O\left(\frac{nr\phi^{1/3}}{p^{2/3}}\right)$



Predictions

- When $\phi = \mathrm{nnz}(S)/nr$ is **low**:
  - Communicating the sparse matrix is cheaper
  - 1.5D **sparse shifting** and **sparse replicating** algorithms should perform faster

- When ϕ is **high**:
  - Communicating the dense matrix is cheaper
  - 1.5D **dense shifting** and 2.5D **dense replicating** algorithms should perform faster

- For the range of processor counts we consider, 1.5D algorithms usually outperform 2.5D algorithms

- 1.5D communication-eliding FusedMM saves ~30% of overall communication; 2.5D communication-eliding FusedMM saves 20% of overall communication.

Experiments

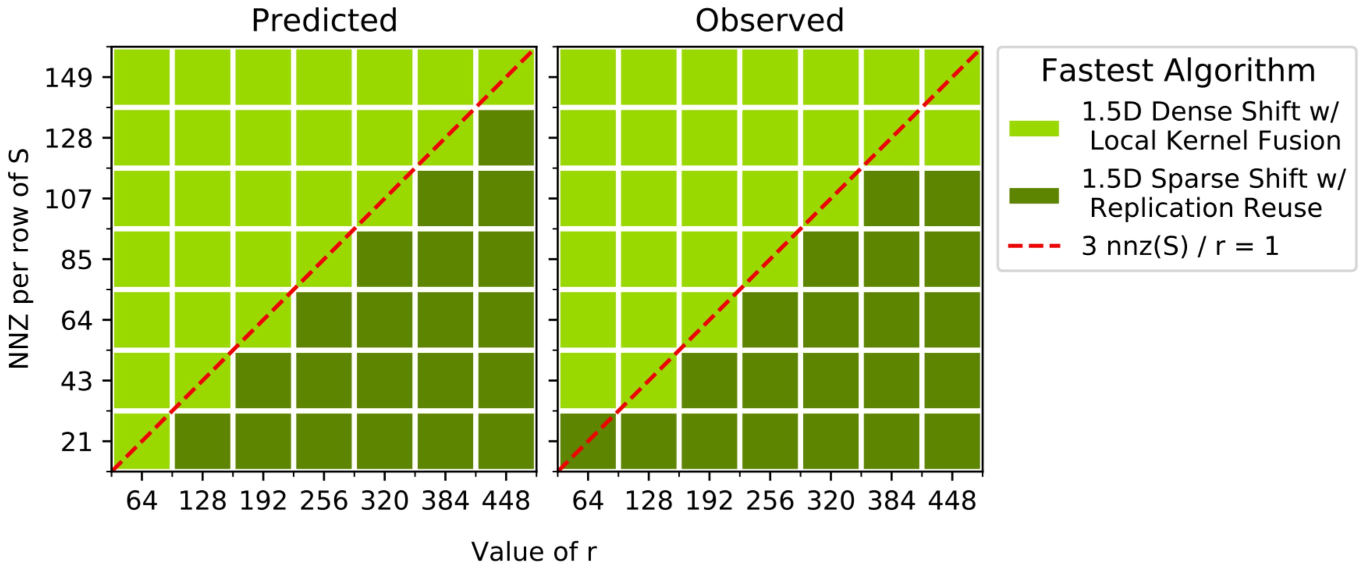
Platform Details

- Experiments run on Cori, a Cray XC40 at Lawrence Berkeley National Laboratory with 256 Xeon Phi Knight Landing (KNL) nodes
- Each node:
  - Has a single CPU with 68 cores
  - Runs at 1.4 GHz
  - Communicates with other nodes via an Aries interconnect arranged using a Dragonfly topology
- We use a hybrid MPI + OpenMP programming model with a single MPI rank and 68 threads node

Credit: National Energy Research Scientific Computing

Performance for Varying ϕ on Erdos-Renyi Matrices

- For $m = n = 2^{22}$ and 32 processors, we vary the nonzero count per row of S and the dense matrix column count r to determine which of our four algorithms performs best



- Prediction closely matches theory: 1.5D dense shifting or 1.5D sparse shifting algorithms are optimal, and the choice between the two depends on the ratio ϕ.

Strong Scaling

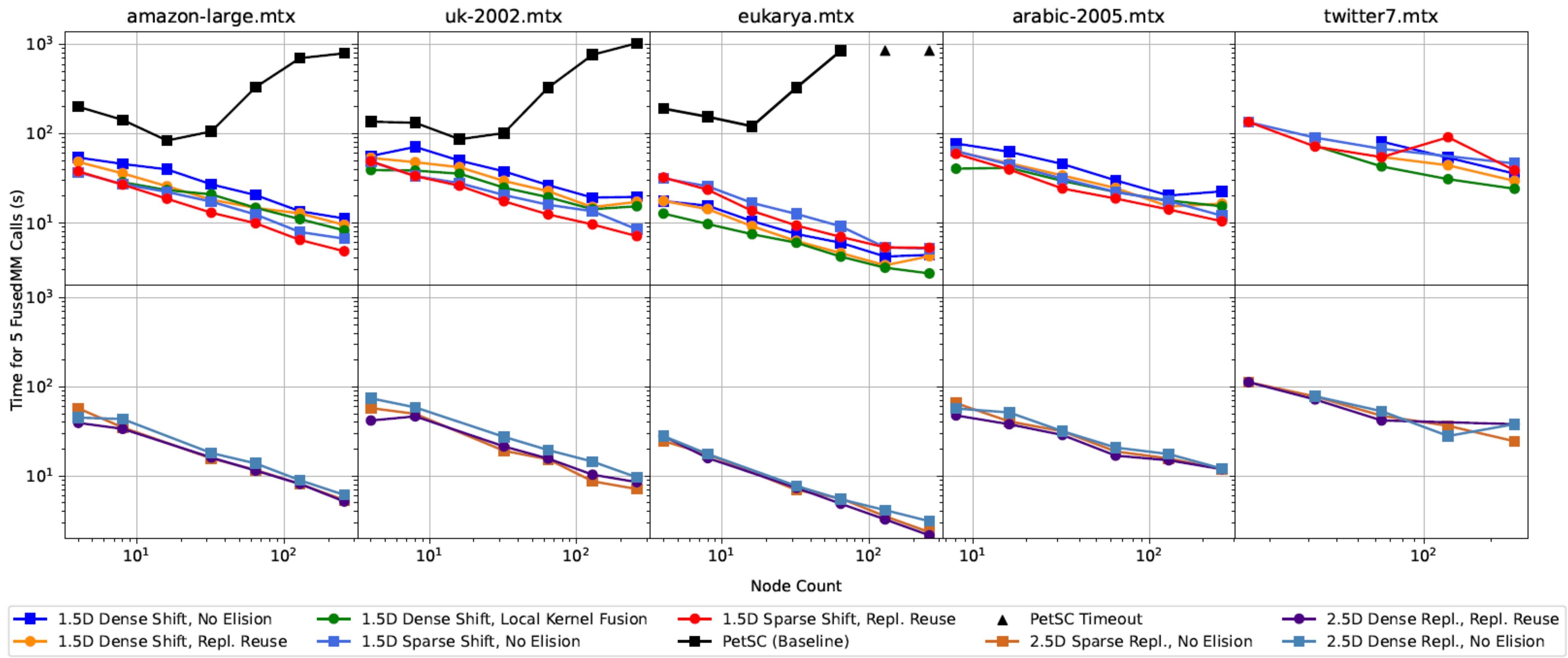
- Compared our FusedMM implementations to two repeated calls of SpMM from the PetSC library (since there is no existing implementation of SDDMM to compare against)

- PetSC only supports 1D partitions of all matrices and does not take advantage of replication. Leads to poor scaling at high processor counts.

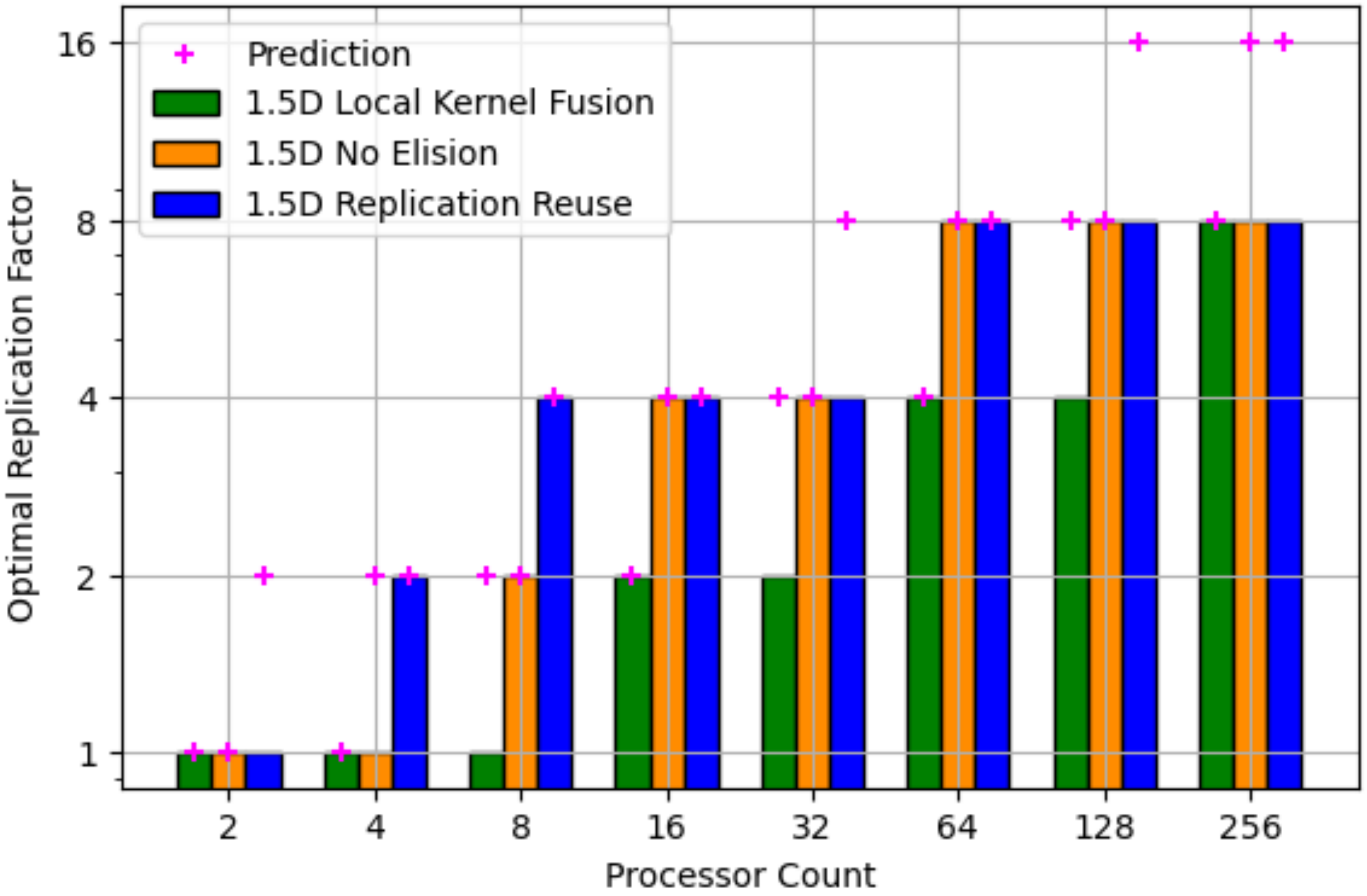
- Algorithms tested on several matrices from the SuiteSparse and a significantly denser matrix from computational biology. $r = 128$ for all experiments

Matrix	Side Length	Nonzero Count	NNZ per Row
amazon-large.mtx	14,249,639	230,788,269	~16
uk-2002.mtx	18,484,117	298,113,672	~16
eukarya.mtx	3,243,106	359,744,161	~111
arabic-2005.mtx	22,744,080	639,999,458	~28
twitter7.mtx	41,652,230	1,468,365,182	~35

Strong Scaling



Predicted vs. Observed Optimal Replication Factor



Application Benchmark 1: Collaborative Filtering

- Netflix-challenge-type computation: compute a low-rank factorization of a sparse matrix $S = A \cdot B^T$ for tall-skinny embedding matrices A, B for the rows and columns.

- Want to minimize squared error norm **only on the nonzero entries** of S

- Idea: **alternately optimize** either A or B, keeping the other matrix fixed. Solve an independent least squares problem $Mx_i = b_i$ for every row i of the unfixed matrix

- Solution: use a Krylov method, conjugate gradients in our case. Use SDDMM / SpMM to compute all query vectors Mx_i in parallel.

Application Benchmark 2: Graph Attention Network

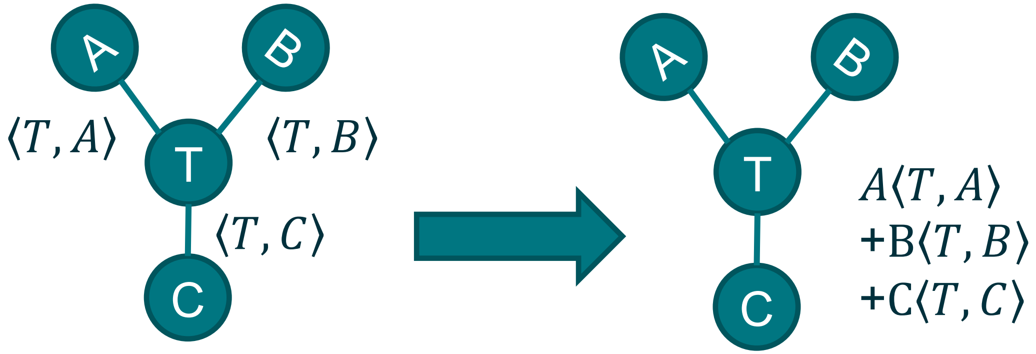
- Graph neural networks learn embeddings for each node of a graph. The key operation at each layer is **graph convolution**, which aggregates embeddings of neighbors of each vertex onto that vertex.

- A **single-head GATN** weights each edge by some function of the incident vertex embeddings. Edge weights become coefficients of the aggregation.

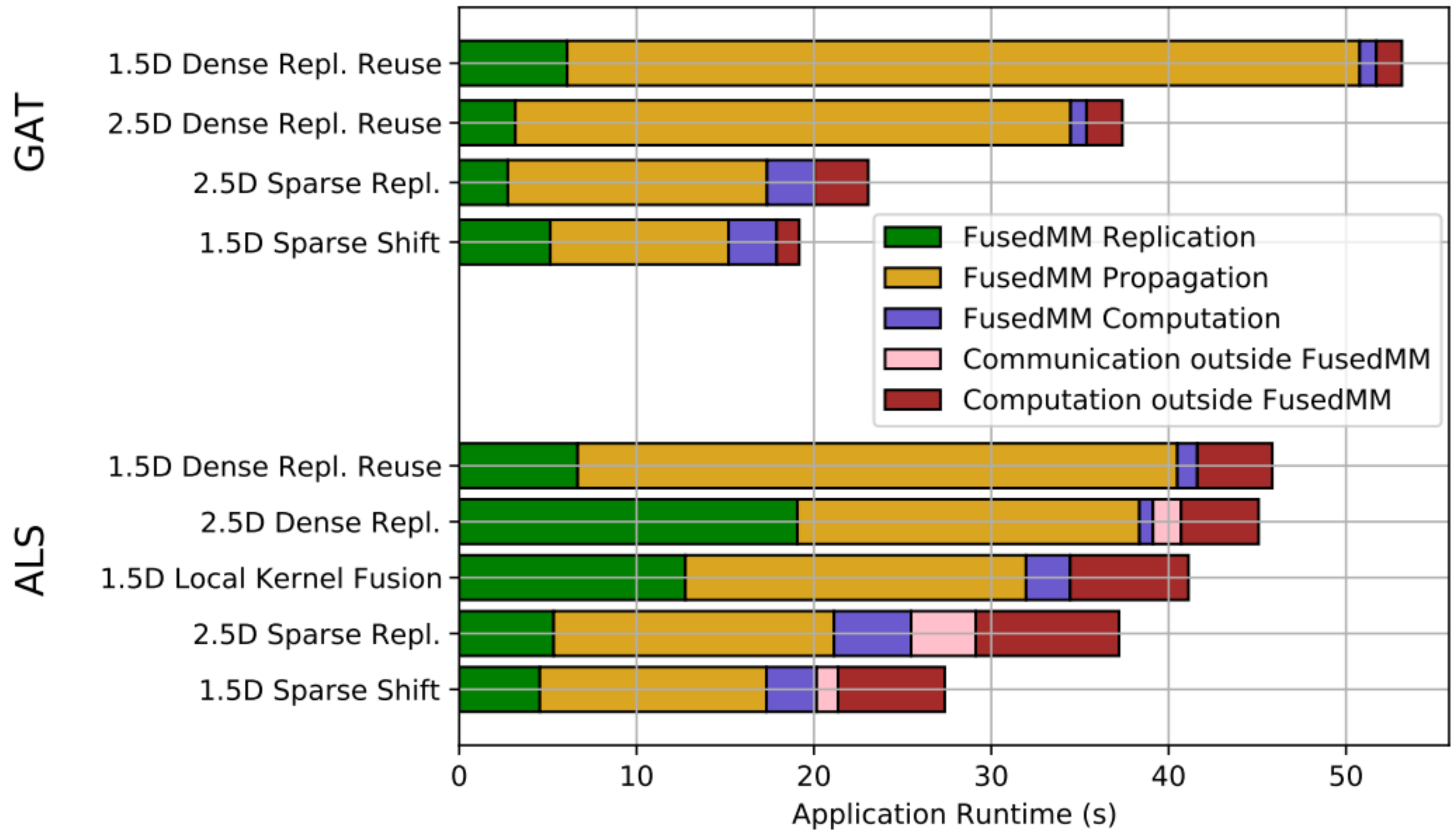
- **Multi-head GATN:** Concatenates the outputs of single heads.

- Message generation / aggregation performed by SDDMM, SpMM respectively.

```
// Input features, features per head,
layers.emplace_back(256, 256, 4);
layers.emplace_back(1024, 256, 4);
layers.emplace_back(1024, 256, 6);
gnn.reset(new GAT(layers, d_ops));
```



Application Performance Breakdown



Summary

- We gave a theoretical communication analysis of sparsity-agnostic communication-avoiding algorithms for SDDMM and FusedMM

- Our algorithms take advantage of extra memory on nodes by replicating inputs, scaling to hundreds of nodes and thousands of cores

- We embedded and tested our algorithms within two applications that use FusedMM

- Further work:
  - More effective overlap between communication and local computation
  - Implementations with one-sided MPI or RDMA
  - Porting implementation to GPUs

Thank you!

Read the paper here

Get the code at github.com/PASSIONLab/distributed_sddmm

Extra Slides

Weak Scaling

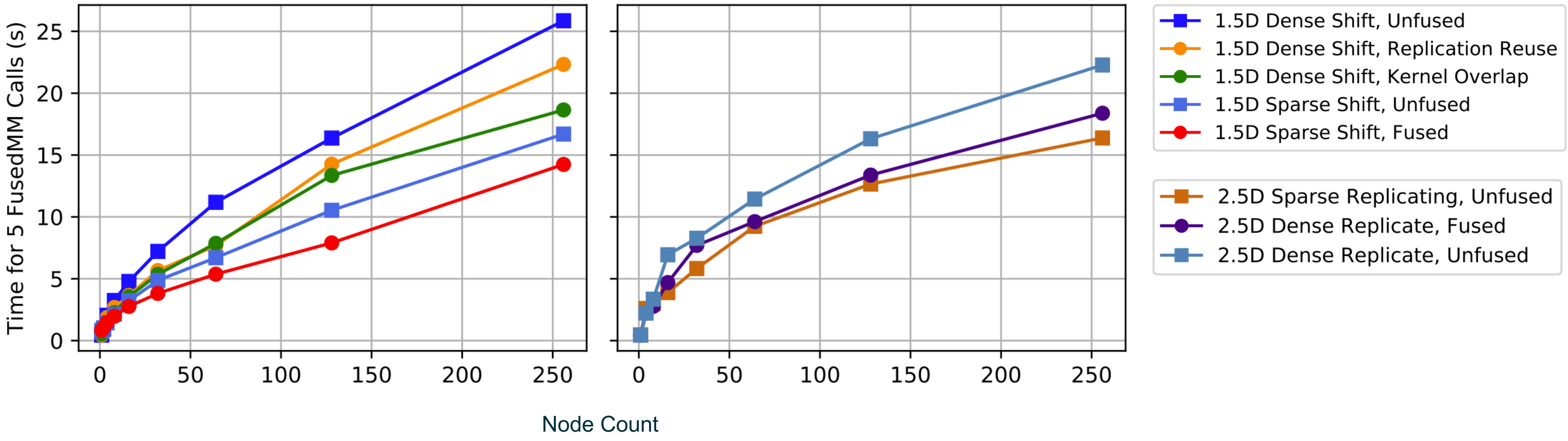
- We examine scaling behavior when keeping the FLOPs per processor constant.
- **Setup 1:**
  - Processor count doubles for each successive experiment
  - The sparse matrix side-length doubles from experiment to experiment
  - The nonzero count per row of the sparse matrix remains constant at 32
  - The embedding dimension r remains constant at 256

- The ratio $\phi = \mathrm{nnz}(S)/nr$ remains constant

- The fraction of nonzeros in the sparse matrix successively decays by a factor of 2

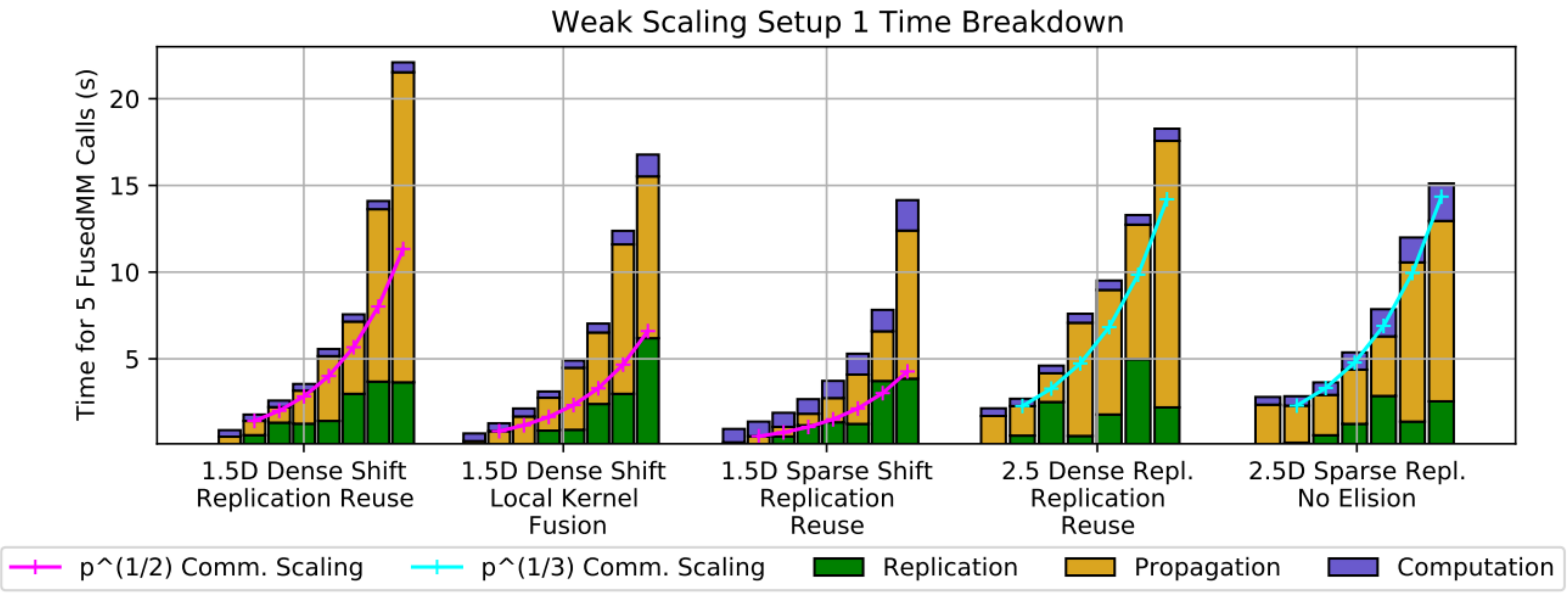
- We expect $p^{1/2}$ communication scaling 1.5D algorithms and $p^{1/3}$ scaling for the 2.5D algorithms

Weak Scaling



- Both local kernel fusion and replication reuse yield communication savings. Local kernel fusion tends to outperform replication reuse
  - Broadcast collective disproportionately expensive at higher processor counts

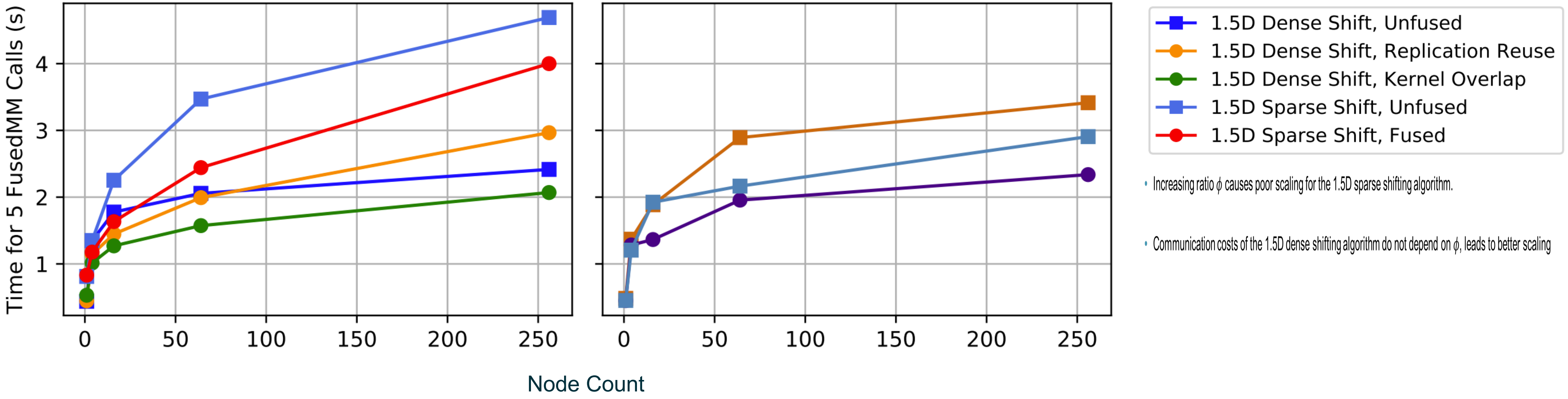
Weak Scaling: Setup 1 Performance Breakdown



Weak Scaling: Setup 2

- Setup 2: For each successive experiment,
  - Processor count quadruples
  - The sparse matrix side length doubles
  - The nonzero count per row of the sparse matrix doubles with an initial value of 32
  - The embedding dimension r remains constant at 256
- The ratio $\phi = \mathrm{nnz}(S)/nr$ successively doubles
- The fraction of nonzeros in the sparse matrix remains constant
- We expect communication to stay constant for 1.5D dense shifting algorithms and even decrease for the 2.5D algorithms. Unlikely in practice due to decreasing node locality.

Weak Scaling: Setup 2



- Increasing ratio ϕ causes poor scaling for the 1.5D sparse shifting algorithm.
- Communication costs of the 1.5D dense shifting algorithm do not depend on ϕ, leads to better scaling

Sparsity-Aware vs. Sparsity-Agnostic SpMM

- We categorize existing SpMM algorithms as either sparsity-aware or sparsity-agnostic
- Sparsity-aware algorithms divide the dense and sparse matrices evenly among processors. If a processor does not own an embedding it needs to process a nonzero, it fetches the embedding from the owning processor
- Communication Cost: Modelled by the edge cut metric of a hypergraph partition of the sparse matrix
- These methods benefit from graph / hypergraph partitioning to reorder nonzeros

	v_1	v_2	v_3
e_1	0	0	1
e_2	1	0	1
e_3	0	1	1
e_4	1	1	0
e_5	0	1	0

Hypergraph Partition into 2 Components of a Sparse Matrix

EKM1 Metric: 2