PARALLEL DENSE MATRIX MULTIPLICATION

Comparing Three Parallel Strategies using MPI

Erfan Habibi Panah Fard

CSE 633 – Parallel Algorithms | Spring 2026

University at Buffalo The State University of New York

Outline

- What is Dense Matrix Multiplication?
- Why parallelize? Strong vs Weak scaling
- Sequential algorithm and complexity
- How to split the work: 1D vs 2D
- Algorithms: 1D Ring, Cannon's, Fox's (with examples)
- Non-blocking communication for all 3
- Experimental setup and methodology
- Multi-node scaling (1, 2, and 4 nodes)
- Results: Strong scaling and Weak scaling
- Key findings and future work

What is Dense Matrix Multiplication?

- Given two square matrices A and B, both of size N × N
- Compute C = A × B, where each element:
- **$C[i][j] = \Sigma\ A[i][k] \times B[k][j]$ (sum over k)**
- "Dense" means most entries are nonzero
- Every element participates in the computation
- Complexity: $O(N^3)$ — for N=2048, that is 8.5 billion operations

Why Parallelize?

- Matrix multiplication is used everywhere:
  - Scientific computing: weather simulation, physics
  - Machine learning: neural network training
  - Linear algebra: solvers, eigenvalue decomposition
- The sequential algorithm is $O(N^3)$ — it gets very slow for large N
- My serial baseline: N=2048 takes about 33.5 seconds on one core
- With 64 processes, I reduced it to 0.44 seconds — 76 times faster!
- Goal: split the matrix across P processes and use MPI to communicate

Sequential Algorithm: Triple Nested Loop

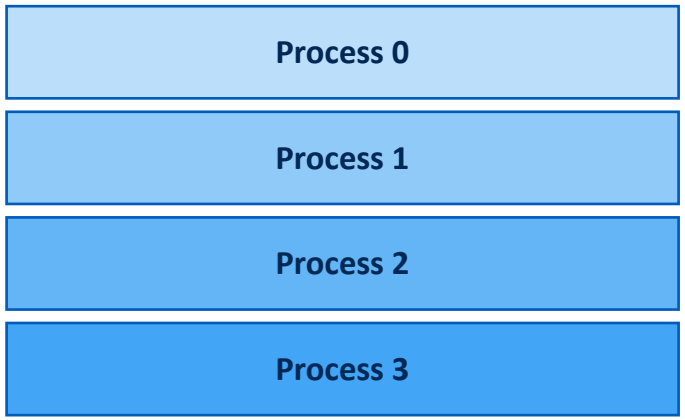
```
for i = 0  to  N - 1 :
   for j = 0  to  N - 1 :
      for k = 0  to  N - 1 :
         C[i][j]  +=  A[i][k] × B[k][j]
```

Complexity	$O(N^3)$ $2N^3$ floating-point ops
Memory	$3 \times N^2$ elements (matrices A, B, C)
Bottleneck	Cache misses for large N
My Baseline	N=2048: 33.5 seconds

Each C[i][j] needs N multiplications × N^2 total elements = N^3 operations

How to Split the Work: 1D vs 2D

1D Row Decomposition



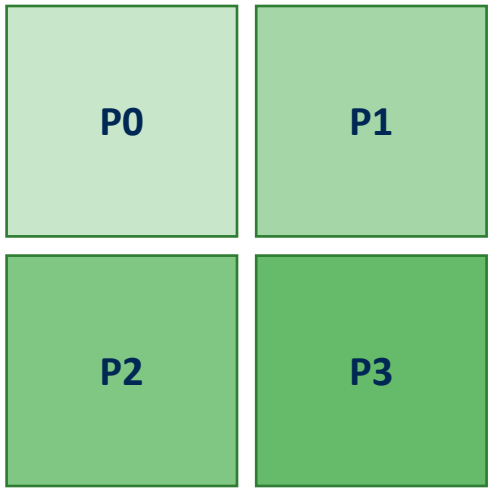
Each process gets N/P rows

Any value of P works

Comm: $O(N^2 \cdot P)$

Simple but communication grows with P

2D Block Decomposition



$\sqrt{P} \times \sqrt{P}$ grid of blocks

P must be a perfect square

Comm: $O(N^2/\sqrt{P})$

Less communication — scales much better!

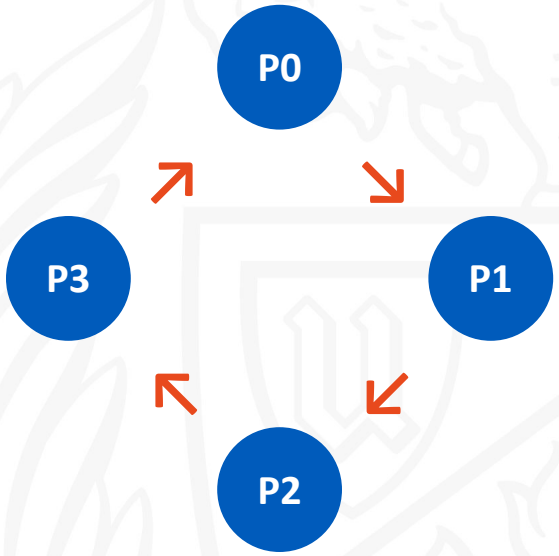
Algorithm 1: 1D Ring-Shift

1. **Distribute**
   Each process gets N/P rows of A and C, plus local B block

2. **Compute**
   Multiply local A rows with current B block → partial C

3. **Shift B**
   Send B block to next neighbor in the ring

4. **Repeat**
   After P-1 shifts, all processes have seen all B rows



B rows rotate clockwise

Comm: $O(N^2 \cdot P)$ Comp: $O(N^3/P)$

1D Ring-Shift — Example with 4 Processes

Setup: 4×4 matrix, P = 4 processes

A rows
(stay fixed)

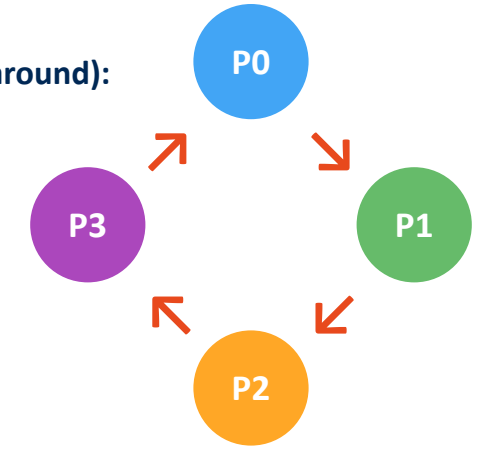
- P0: [1, 2, 3, 4]
- P1: [5, 6, 7, 8]
- P2: [9, 10, 11, 12]
- P3: [13, 14, 15, 16]

B rows
(rotate!)

→

- B row 0: [a, b, c, d]
- B row 1: [e, f, g, h]
- B row 2: [i, j, k, l]
- B row 3: [m, n, o, p]

The ring (B rows pass around):



B rows pass clockwise

The 4 Rounds:

1. **Round 1**
   Multiply A row × B row
   Then pass B to next
2. **Round 2**
   Got a new B row
   Multiply again, pass
3. **Round 3**
   Another new B row
   Multiply again, pass
4. **Round 4**
   Last B row arrives
   Multiply — Done! ✓

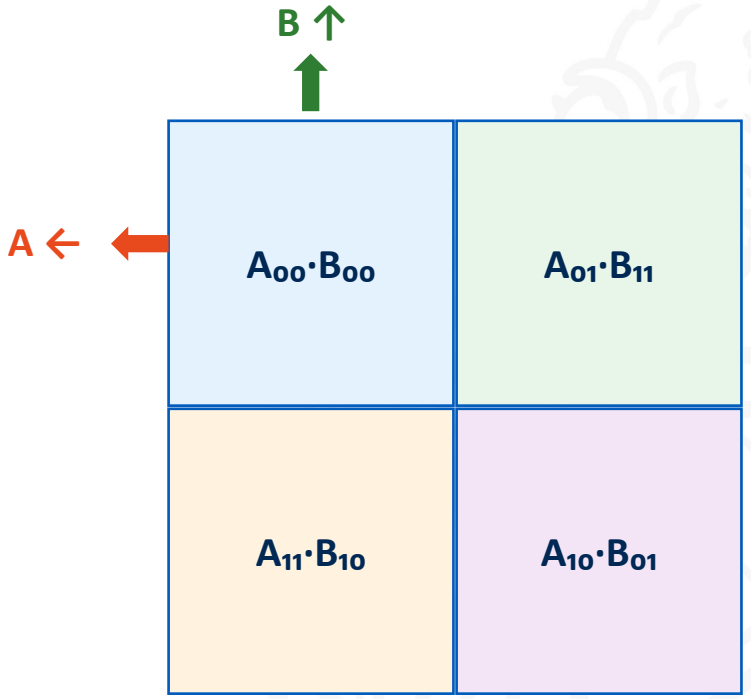
Why does this work?

To compute C[i][j], process i needs its A row (already has it) and ALL rows of B.

After 4 shifts, every process has seen every B row → result is complete!

Algorithm 2: Cannon's 2D Algorithm

1. **Align**
   Shift row i of A left by i
   Shift col j of B up by j
2. **Multiply**
   C_local += A_local × B_local
3. **Shift**
   Shift all A blocks left by 1
   Shift all B blocks up by 1
4. **Repeat**
   Steps 2-3 for √P iterations total



After initial alignment

Key advantage: Only nearest-neighbor shifts — Comm $O(N^2/\sqrt{P})$ scales much better than 1D

Cannon's 2D — Example with 4 Processes

Before (original blocks):

P0: A_{00}, B_{00}	P1: A_{01}, B_{01}
P2: A_{10}, B_{10}	P3: A_{11}, B_{11}

align!

After alignment:

col 1 of B: shift up by 1

P0: A_{01}, B_{10}	P1: A_{00}, B_{11}
P2: A_{11}, B_{00}	P3: A_{10}, B_{01}

row 1 of A: shift left by 1

Then 2 rounds (√P = 2):

1 Multiply local blocks:

P0: $C_{00} \mathrel{+}= A_{00} \times B_{00}$ ✓
P1: $C_{01} \mathrel{+}= A_{01} \times B_{11}$ ✓
P2: $C_{10} \mathrel{+}= A_{11} \times B_{10}$ ✓
P3: $C_{11} \mathrel{+}= A_{10} \times B_{01}$ ✓

Then shift A ← 1, shift B ↑ 1

2 New blocks, multiply again:

P0: $C_{00} \mathrel{+}= A_{01} \times B_{10}$ ✓
P1: $C_{01} \mathrel{+}= A_{00} \times B_{01}$ ✓
P2: $C_{10} \mathrel{+}= A_{10} \times B_{00}$ ✓
P3: $C_{11} \mathrel{+}= A_{11} \times B_{11}$ ✓

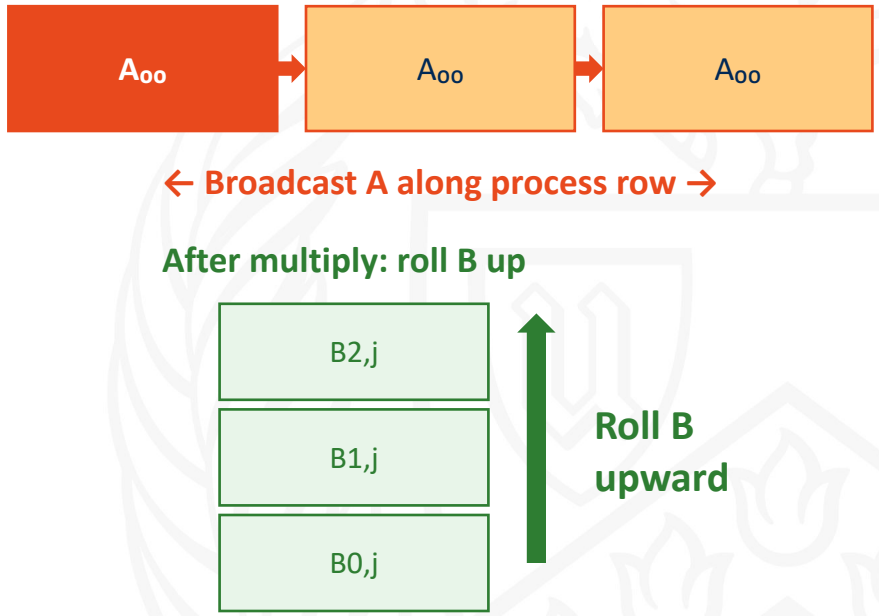
Done! Every C block is complete.

Why the alignment trick?

$C_{00} = A_{00} \times B_{00} + A_{01} \times B_{10}$. After alignment, P0 has exactly A_{00} and B_{00} for round 1, then after shifting it gets A_{01} and B_{10} for round 2. Every pair matches up!

Algorithm 3: Fox's 2D (Broadcast-Multiply-Roll)

1. **Broadcast**
   At step k, process (i,j) broadcasts A[i, (i+k) mod √P] across row i
2. **Multiply**
   C_local += A_broadcast × B_local
3. **Roll B**
   Shift B block up by 1 position
4. **Repeat**
   For √P total iterations



vs Cannon: Broadcast replaces initial alignment — same $O(N^2/\sqrt{P})$ comm, different pattern

Fox's 2D — Example with 4 Processes

Start (no alignment needed!):

P0: A_{00}, B_{00}	P1: A_{01}, B_{01}
P2: A_{10}, B_{10}	P3: A_{11}, B_{11}

Round 1:

① **Broadcast diagonal A:**

A_{00} → A_{00} P0 & P1 get A_{00}

A_{11} ← A_{11} P2 & P3 get A_{11}

② **Multiply:**

P0: C_{00} += A_{00}×B_{00} ✓
P1: C_{01} += A_{00}×B_{01} ✓
P2: C_{10} += A_{11}×B_{10} ✓
P3: C_{11} += A_{11}×B_{11} ✓

③ **Roll B up by 1**

P0←B_{10} P1←B_{11}
P2←B_{00} P3←B_{01}

Round 2:

① **Broadcast next A:**

A_{01} → A_{01} P0 & P1 get A_{01}

A_{10} → A_{10} P2 & P3 get A_{10}

② **Multiply:**

P0: C_{00} += A_{01}×B_{10} ✓
P1: C_{01} += A_{01}×B_{11} ✓
P2: C_{10} += A_{10}×B_{00} ✓
P3: C_{11} += A_{10}×B_{01} ✓

Done! All C blocks complete ✓

Fox vs Cannon — same result, different communication:

Cannon: alignment + shift A left + shift B up (neighbor only)

Fox: no alignment, broadcast A along row + roll B up

Algorithm Comparison Summary

	1D Ring-Shift	Cannon's 2D	Fox's 2D
Decomposition	1D — row blocks	2D — square blocks	2D — square blocks
Process count	Any P	P must be perfect square	P must be perfect square
Communication pattern	Ring shift of B rows to neighbor	Shift A left, shift B up	Broadcast A in row, roll B up
Comm volume per process	$O(N^2)$	$O(N^2 / P)$	$O(N^2 / \sqrt{P})$
Best for	Simple cases, flexible P	Large P, low bandwidth systems	Large P, latency-sensitive systems

Experimental Setup

Hardware — UB CCR Cluster

CPU: Intel Xeon Gold 6548Y+

Cores: 64 per node (2 × 32-core sockets)

Memory: 503 GB per node

Experiment Parameters

Matrix size: N = 2048

Operations: 8.59 billion operations

Processes: P = 1, 2, 4, 8, 16, 32, 64

Strong scaling: Fix N = 2048, increase P

Weak scaling: Fix 256 rows per process

All results verified for correctness · Timing with MPI_Wtime · Excludes matrix generation

Why Non-blocking Communication?

Blocking

Compute	Wait + Send
Compute	Wait + Send

CPU waits during data transfer

Computation pauses

Result: idle CPU time

Non-blocking (overlap)

Compute (current data)
Send next data (in background)

Start transfer, return immediately

Compute while data moves

Result: better CPU utilization

MPI calls: MPI_Isend, MPI_Irecv, MPI_Wait

Multi-node Setup on CCR

1 Node

Up to P=64
(64 cores)

2 Nodes

Node 1
64 cores

Node 2
64 cores

P=128 total

4 Nodes

N1 64c | N2 64c

N3 64c | N4 64c

P=256 total

Experimental Methodology

Why multiple runs?

Single runs can have:

- System load spikes
- OS interrupts
- Network glitches
- Cache state variations

Solution: Run each config many times

Report avg + min/max

10 runs catches outliers

and shows real variance

My setup

10 runs per config (5 for slow N=8192)

All averages reported with min/max range

Error bars on all graphs

--exclusive SLURM flag

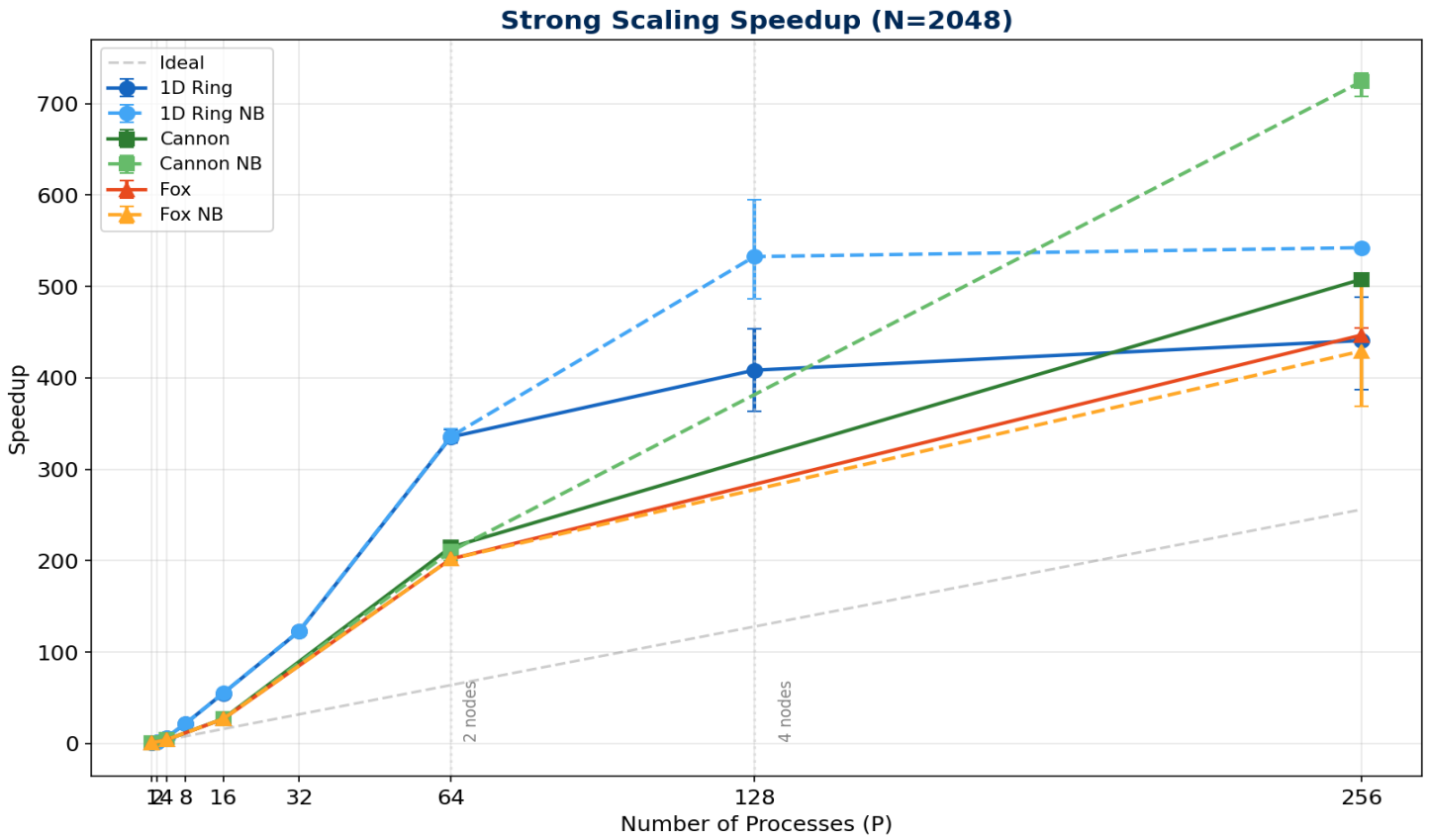
→ No other users on nodes

→ Clean, repeatable timing

Total: ~1500 individual measurements

across 158 result files

Strong Scaling Speedup (N=2048)



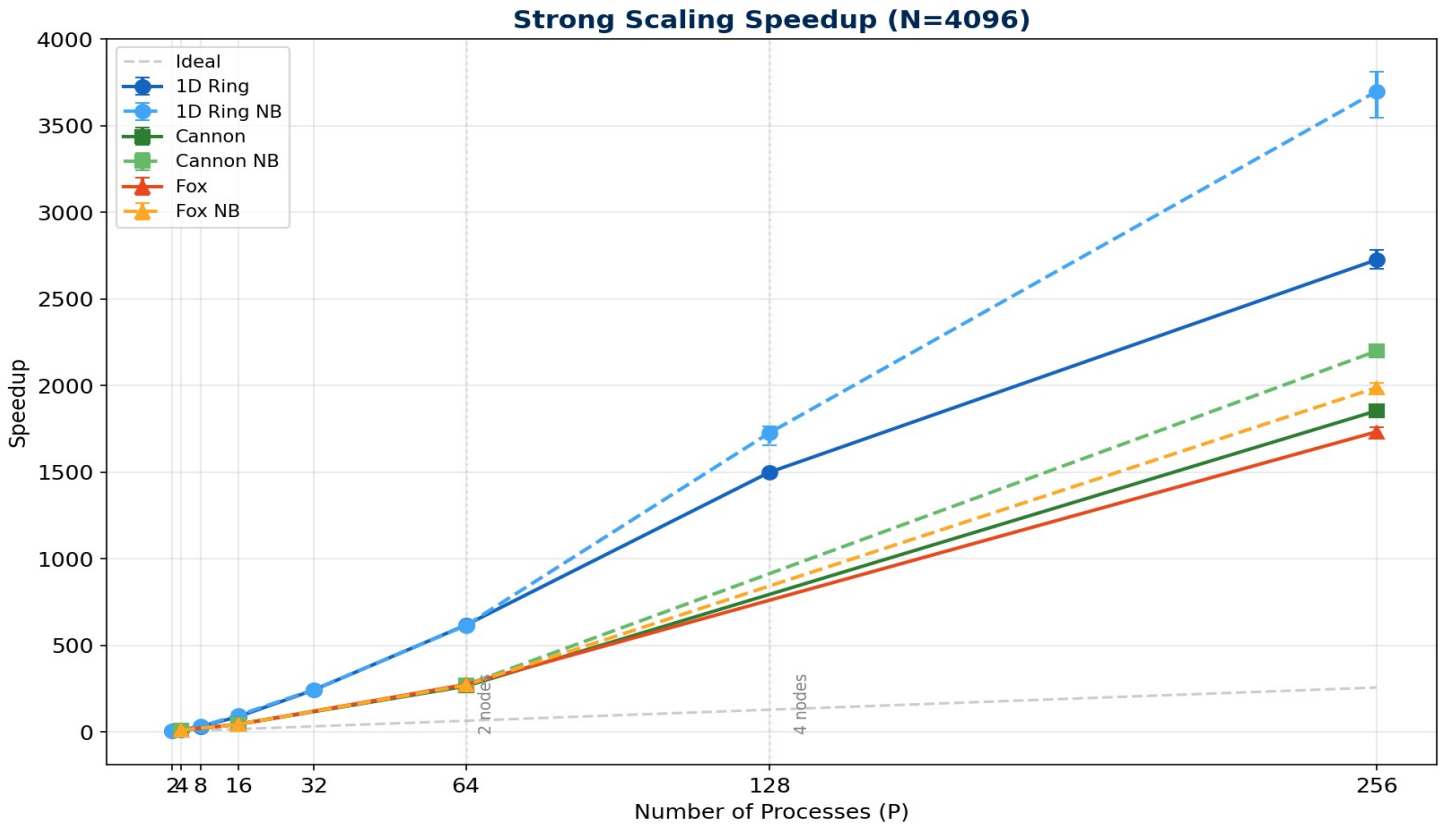
Best Speedups

P=64
336× speedup
(1D Ring NB)
1 node

P=128
533× speedup
(1D Ring NB)
2 nodes

P=256
724× speedup
(Cannon NB)
4 nodes

Strong Scaling Speedup (N=4096)



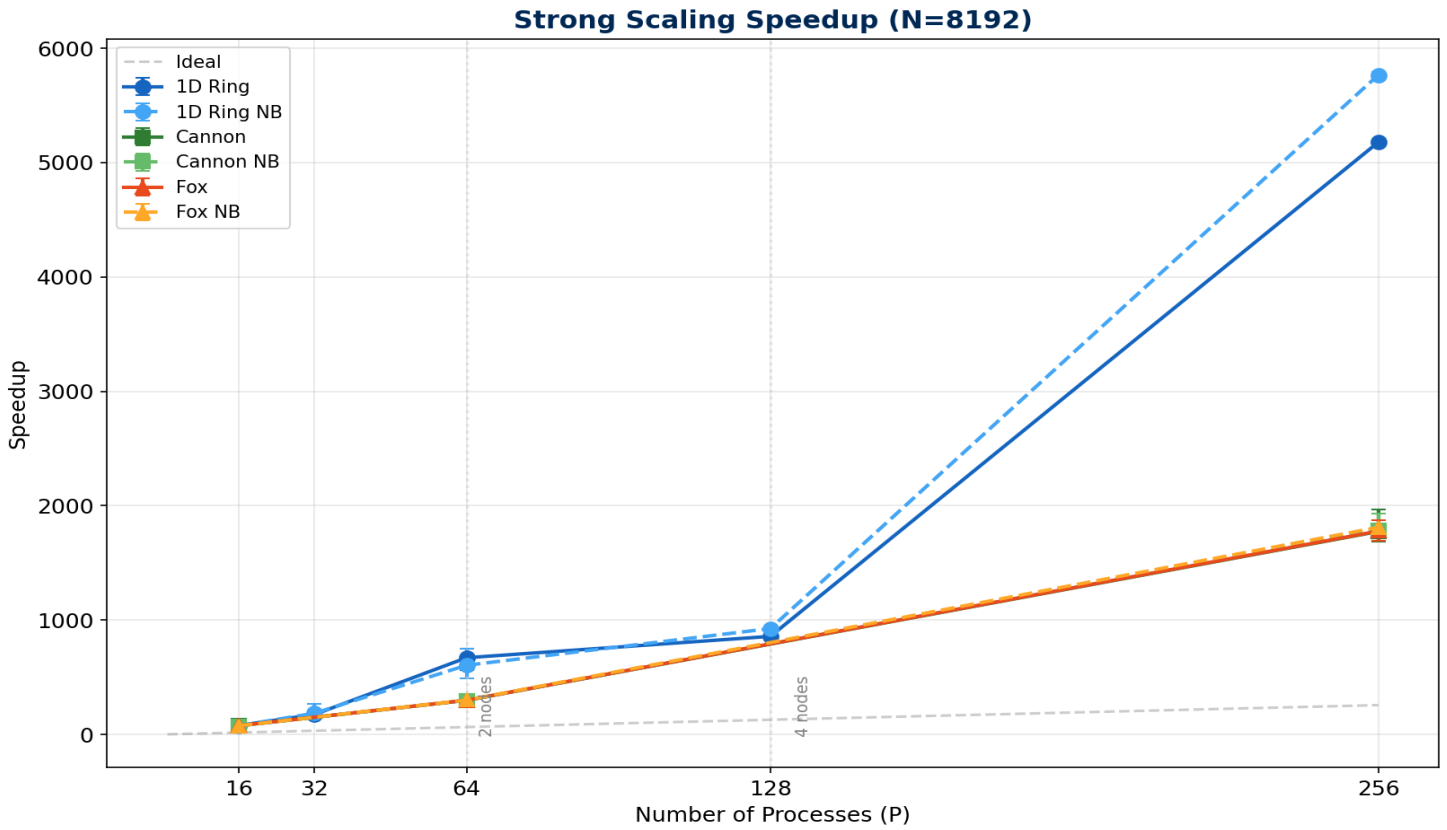
Best Speedups

P=64
616× speedup
(1D Ring)
1 node

P=128
1726× speedup
(1D Ring NB)
2 nodes

P=256
3696× speedup
(1D Ring NB)
4 nodes

Strong Scaling Speedup (N=8192)



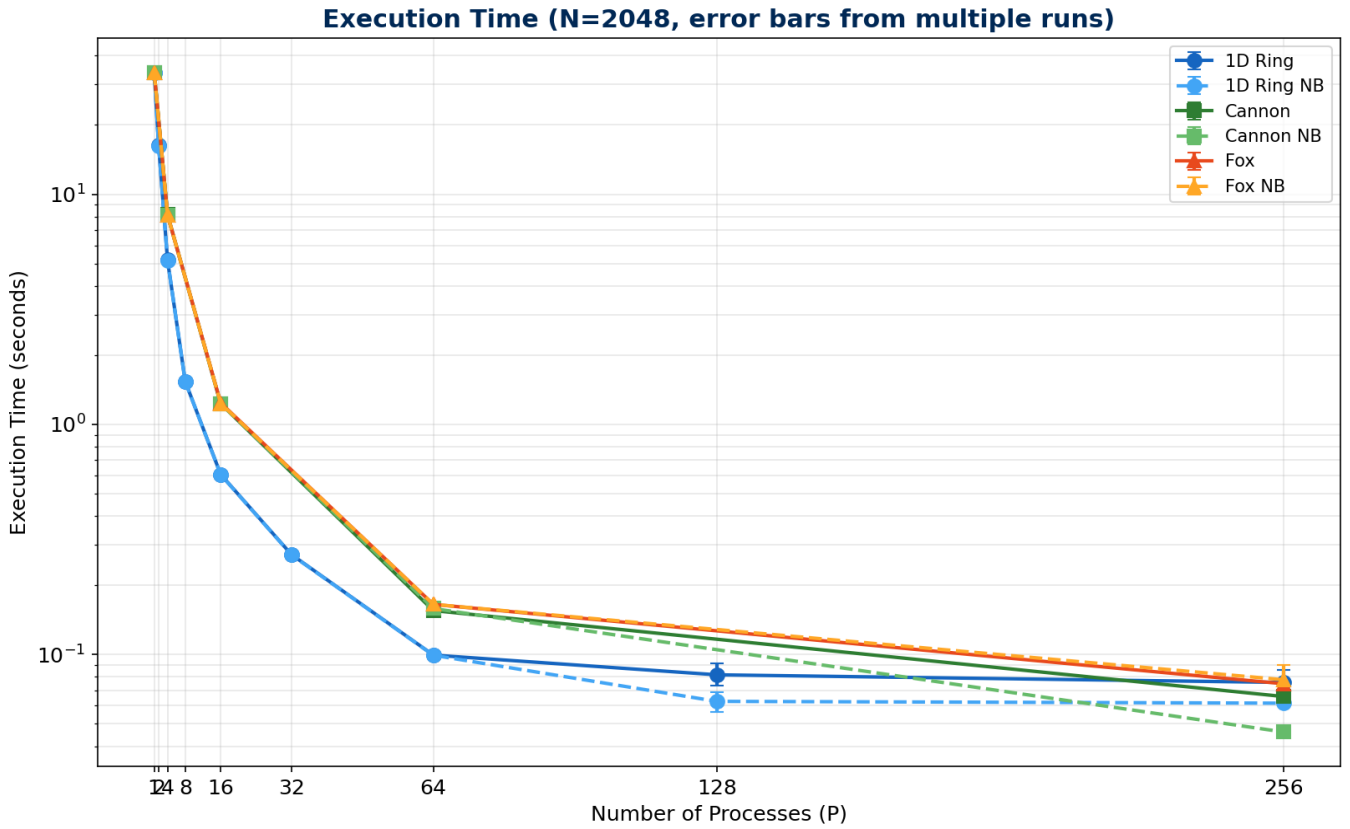
Best Speedups

P=64
670× speedup
(1D Ring)
1 node

P=128
926× speedup
(1D Ring NB)
2 nodes

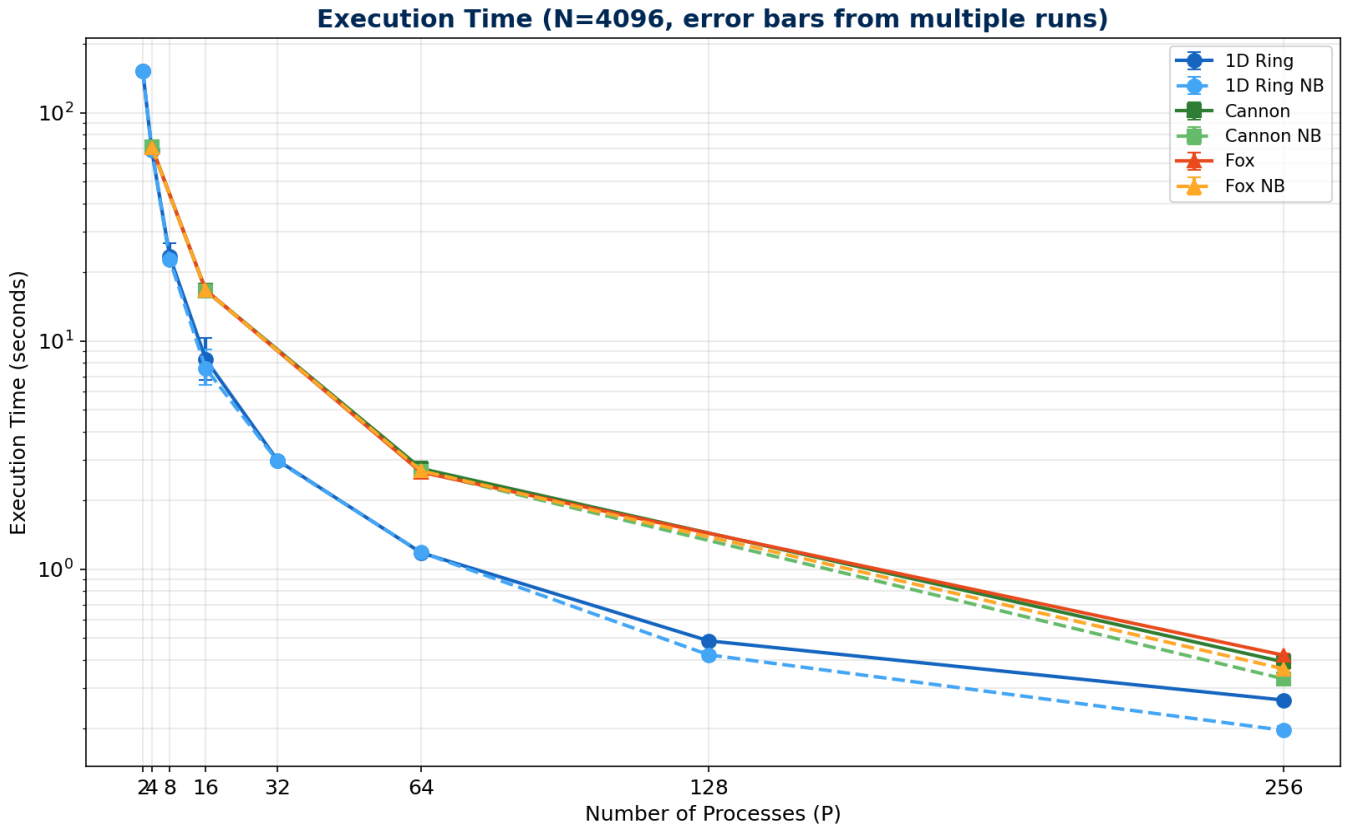
P=256
5765× speedup
(1D Ring NB)
4 nodes

Strong Scaling Times (N=2048)



P	1D Ring	Cannon	Fox
1	33.831	33.818	33.803
4	5.177	8.152	8.140
16	0.606	1.230	1.240
64	0.099	0.155	0.165
256	0.076	0.066	0.075

Strong Scaling Times (N=4096)

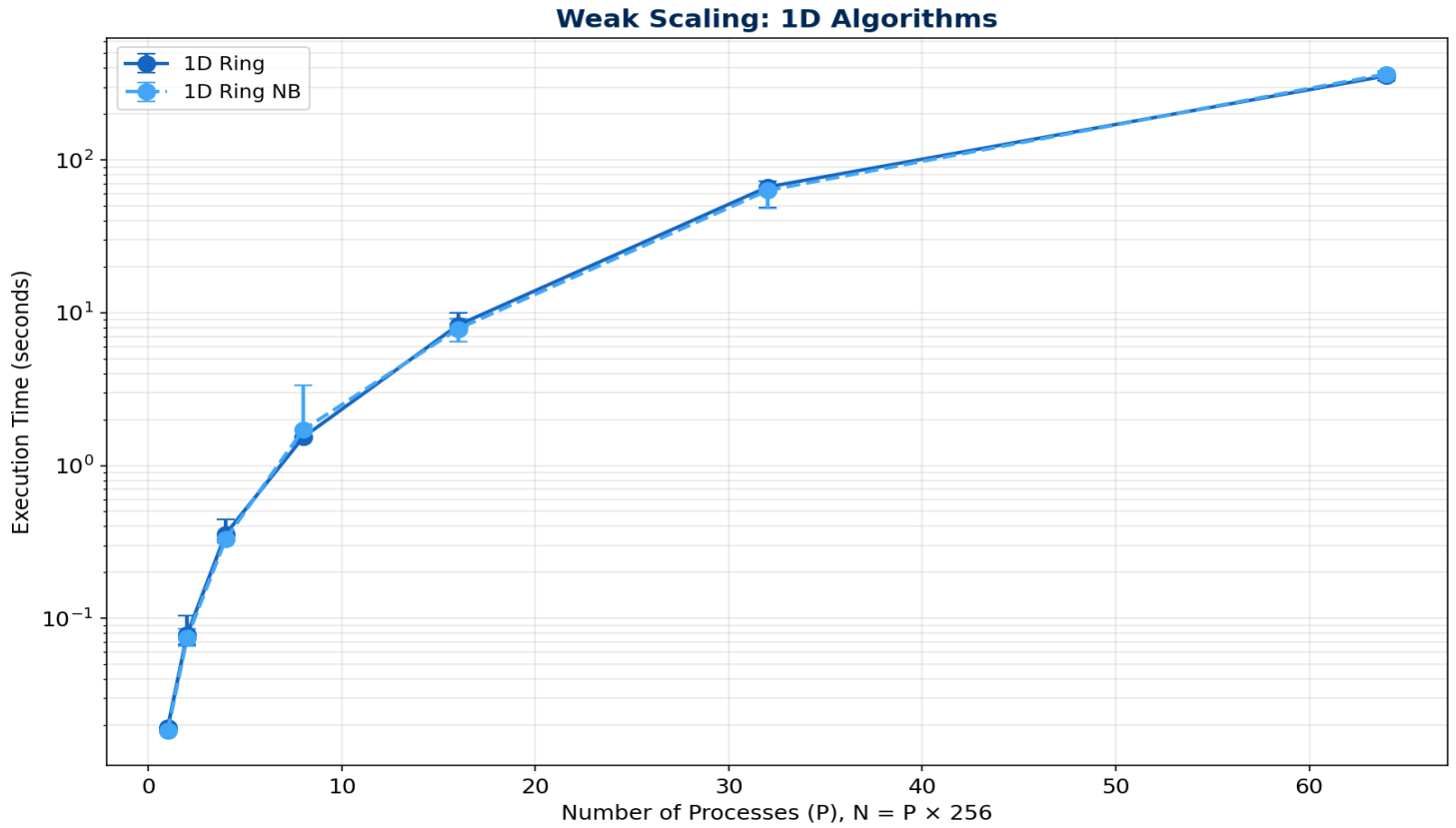


P	1D Ring	Cannon	Fox
4	68.313	70.725	70.603
16	8.300	16.683	16.706
64	1.179	2.757	2.658
256	0.267	0.392	0.420

Strong Scaling Times (N=8192)

P	1D Ring	Cannon	Fox
16	148.430	151.593	151.622
64	17.234	38.790	38.654
256	2.232	6.514	6.501

Weak Scaling: 1D Algorithms (Gustafson)



Setup

Fix: 256 rows per process

N grows with P:

P=1 → N=256

P=4 → N=1024

P=64 → N=16384

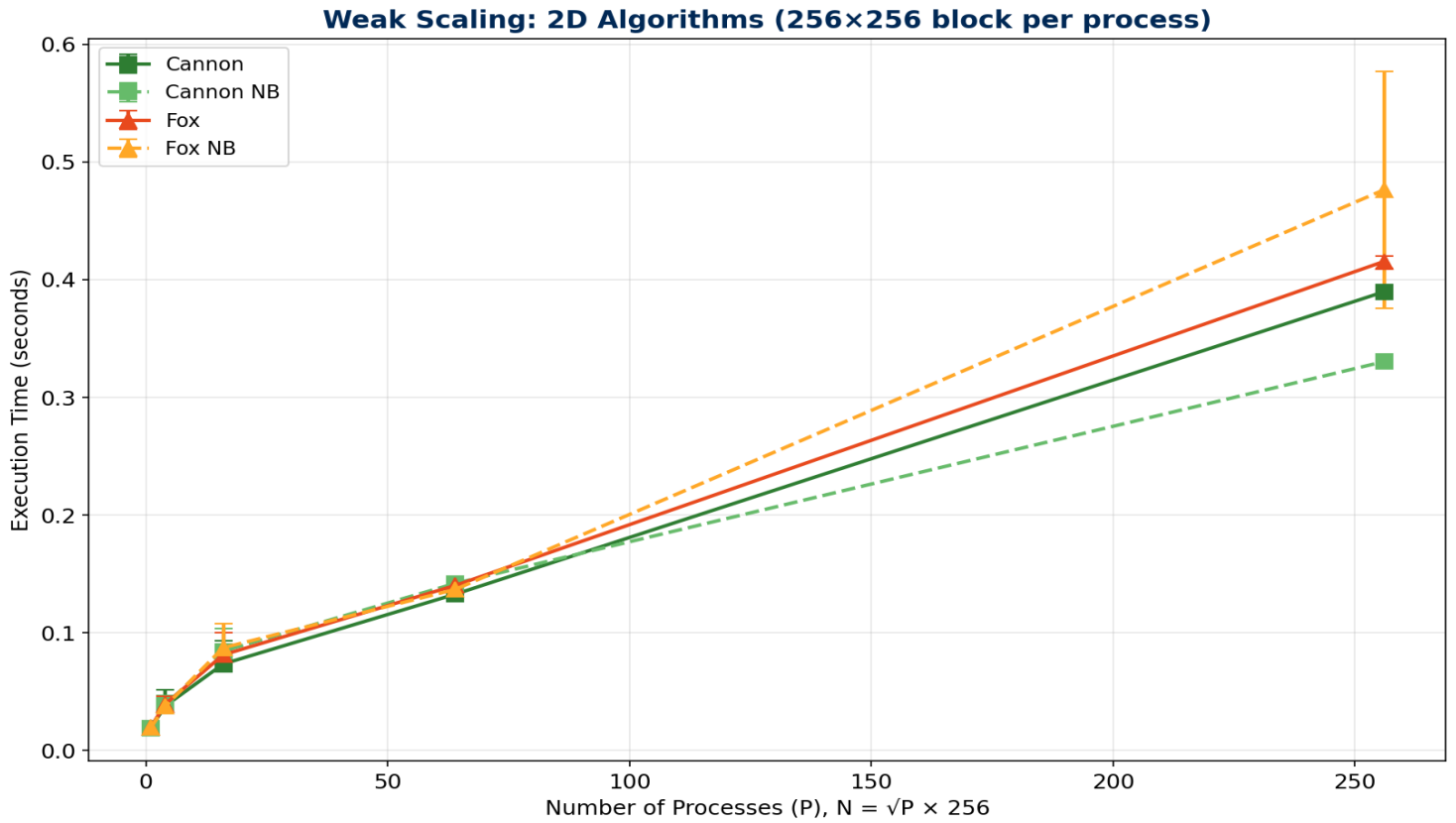
Ideal: flat line

Reality: grows fast

Why?

- Work is $O(N^3)$
- N grows linearly
- Total work $O(P^3)$

Weak Scaling: 2D Algorithms (Gustafson)



Setup

Fix: 256×256 block

N grows with √P:

P=1 → N=256

P=16 → N=1024

P=256 → N=4096

Result: almost flat!

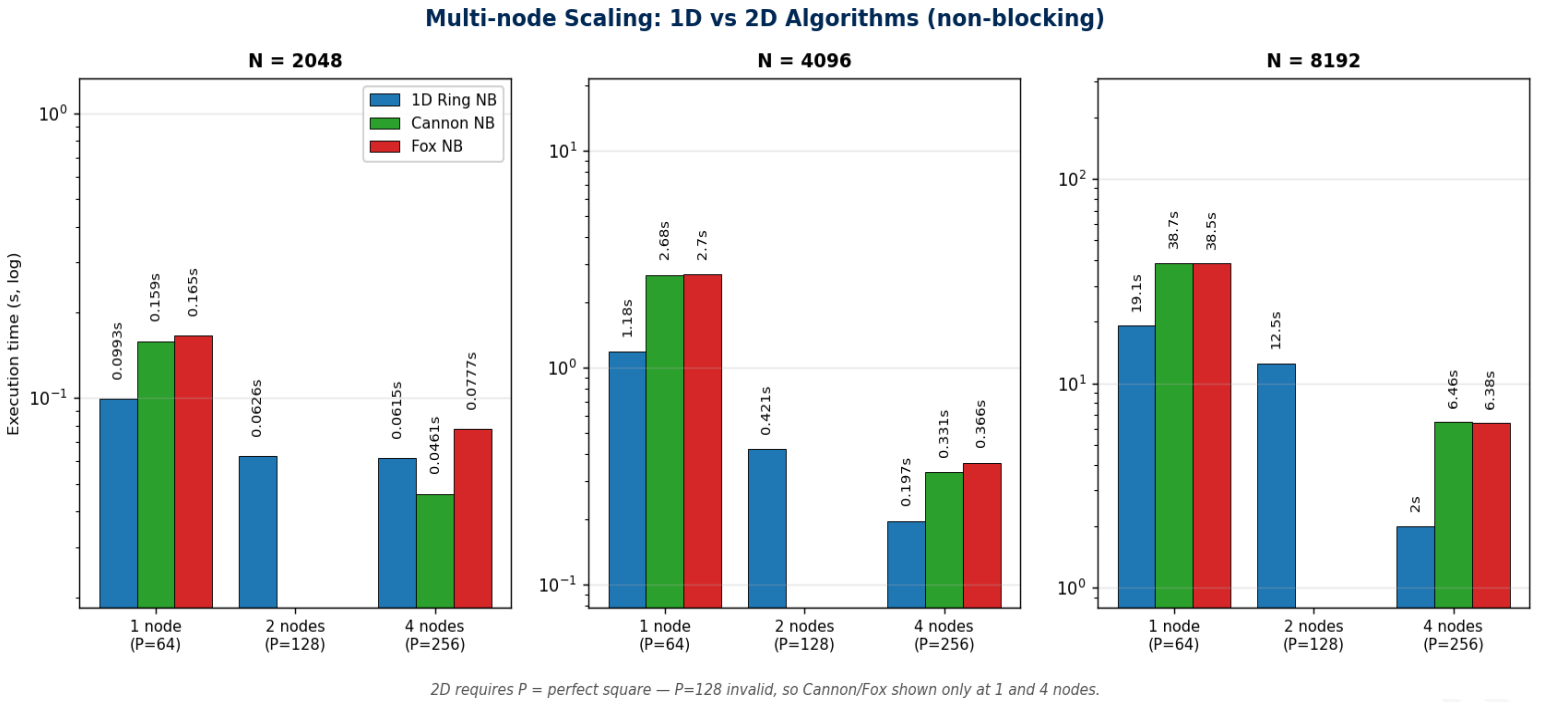
P=1: 0.02s

P=256: 0.39s

Only 20× more time

for 256× more processes

Multi-node Scaling: 1, 2, 4 Nodes



2D requires P = perfect square — P=128 invalid, so Cannon/Fox shown only at 1 and 4 nodes.

Star Result

3696×
speedup

1D Ring NB at P=256
(4 nodes, N=4096)

Serial: 12 min
Parallel: 0.20 sec

That's 3600× faster
than serial!

Cannon and Fox shown at 1 and 4 nodes only — 2D requires P = perfect square; P=128 (2 nodes) is invalid.

All 6 Algorithms — Best Time at Each Size

Algorithm	N=2048 (P=256)	N=4096 (P=256)	N=8192 (P=256)
1D Ring	0.076s (441×)	0.267s (2724×)	2.232s (5177×)
1D Ring NB	0.061s (542×)	0.197s (3696×)	2.004s (5765×)
Cannon	0.066s (508×)	0.392s (1853×)	6.514s (1774×)
Cannon NB	0.046s (724×)	0.331s (2198×)	6.459s (1789×)
Fox	0.075s (447×)	0.420s (1732×)	6.501s (1777×)
Fox NB	0.078s (429×)	0.366s (1987×)	6.381s (1811×)

All times averaged over 5-10 runs. P=256 = 4 nodes × 64 cores.

Key Findings

- Best speedup: 3696× for 1D Ring NB at P=256, N=4096 (4 nodes)
- Multi-node scales well: P=64 → P=128 → P=256 each gives improvement
- Non-blocking helps most at multi-node, less at single node
- 2D weak scaling is excellent: 256× more processes → only 20× more time
- 1D weak scaling shows $O(N^3)$ work growth — known limitation
- All 6 algorithms produce correct results across all configurations

Future Work

- Implement Strassen's algorithm: O(N^2.807) for further reduction
- Test on more nodes (8 or 16) for higher process counts
- Add detailed communication breakdown for all algorithms
- Compare different block sizes for cache optimization
- Try GPU acceleration with CUDA or OpenACC

References

- [1] Golub & Van Loan, Matrix Computations, 4th ed., Johns Hopkins Press, 2013
- [2] Cormen et al., Introduction to Algorithms, 3rd ed., MIT Press, 2009
- [3] Cannon, "A cellular computer for the Kalman filter," PhD thesis, Montana State Univ., 1969
- [4] Fox et al., "Matrix algorithms on a hypercube," Parallel Computing 4(1), 1987
- [5] Grama et al., Introduction to Parallel Computing, 2nd ed., Pearson, 2003
- [6] Quinn, Parallel Programming in C with MPI and OpenMP, McGraw-Hill, 2004
- [7] Gustafson, "Reevaluating Amdahl's Law," Comm. ACM 31(5), 1988
- [8] Amdahl, "Validity of the single processor approach," AFIPS, 1967

THANK YOU

Questions?

Erfan Habibi Panah Fard

CSE 633 – Parallel Algorithms | Spring 2026

University at Buffalo The State University of New York