

Please check the link below to see the programming part of this Project. The repository includes:

Link: <https://github.com/ZJWei2002/AMS530-Projects/tree/main/Project3>

## Project Structure

The Project3 folder contains the following files and directories:

```
Project3/
├── fox_multiply.py           # Fox's algorithm implementation
├── strassen_multiply.py     # Strassen's algorithm implementation
├── combined_experiments.py  # Experiment runner and plotting script
├── Project 3 -- Zijun Wei.pdf # This report
├── results/                 # Generated results directory
│   ├── timing_results.txt   # Timing data for all (P, N) combinations
│   ├── speedup_curves_fox.png # Speedup plots for Fox's algorithm
│   └── speedup_curves_strassen.png # Speedup plots for Strassen's algorithm
```

## Problem 3.1: Parallel Matrix Multiplication

### Problem Description

**Problem 3.1:** Parallel Matrix Multiplication Using Fox's and Strassen's Methods: Design, implement, and analyze Fox's and Strassen's algorithms for multiplying two large  $N \times N$  matrices **A** and **B** with random elements in  $(-1, 1]$ . Use any available parallel computer and interconnection network.

#### Tasks:

1. Briefly describe both algorithms.
2. Implement them on your parallel system.
3. Test performance for:
  - Cores  $P = 2^2, 2^4, 2^6$  (4, 16, 64 cores)
  - Matrix sizes  $N = 2^8, 2^{10}, 2^{12}$  (256, 1024, 4096)
4. Collect timing data for all 9 (P, N) combinations.
5. Plot speedup curves.
6. Discuss performance trends and scalability.

**Deliverables:** Algorithm descriptions, source code, timing results, speedup plots, and a brief performance analysis.

---

# Part 1: Fox's Algorithm

## 1. Algorithm Description

Fox's algorithm (also known as Broadcast-Multiply-Roll or BMR method) is a parallel matrix multiplication algorithm designed for distributed memory systems with a 2D mesh processor topology.

### Key Concepts

- **Decomposition:** An  $N \times N$  matrix is decomposed into  $\sqrt{P} \times \sqrt{P}$  blocks, where  $P$  is the number of processors
- **Processor Grid:** Processors are arranged in a  $\sqrt{P} \times \sqrt{P}$  grid
- **Data Distribution:** Each processor  $(i,j)$  initially stores:
  - $A_{ij}$ : a block of matrix  $A$
  - $B_{ij}$ : a block of matrix  $B$
  - Computes  $C_{ij}$ : a block of result matrix  $C$

### Algorithm Steps

For each step  $k = 0$  to  $\sqrt{P}-1$ :

1. **Broadcast:** Row  $i$  broadcasts  $A[i, (i+k) \bmod \sqrt{P}]$  to all processors in row  $i$
2. **Multiply:** Each processor multiplies the broadcast  $A$  block with its local  $B$  block
3. **Accumulate:** Add the result to the local  $C$  block
4. **Shift:** Circularly shift  $B$  blocks upward in columns (each processor sends  $B$  to the processor above in the same column)

### Communication Pattern

- $\sqrt{P}$  broadcast operations (one per row per step)
- $\sqrt{P}-1$  shift operations (one per step, except last)
- Total:  $O(\sqrt{P})$  communication steps per iteration

### Time Complexity

- **Computation:**  $O(N^3/P)$  - each processor performs  $N^3/P$  operations
  - **Communication:**  $O(N^2/\sqrt{P})$  per step
  - **Overall:**  $O(N^3/P)$  with communication overhead  $O(N^2\sqrt{P})$
-

## 2. Timing Results (from running combined\_experiments.py)

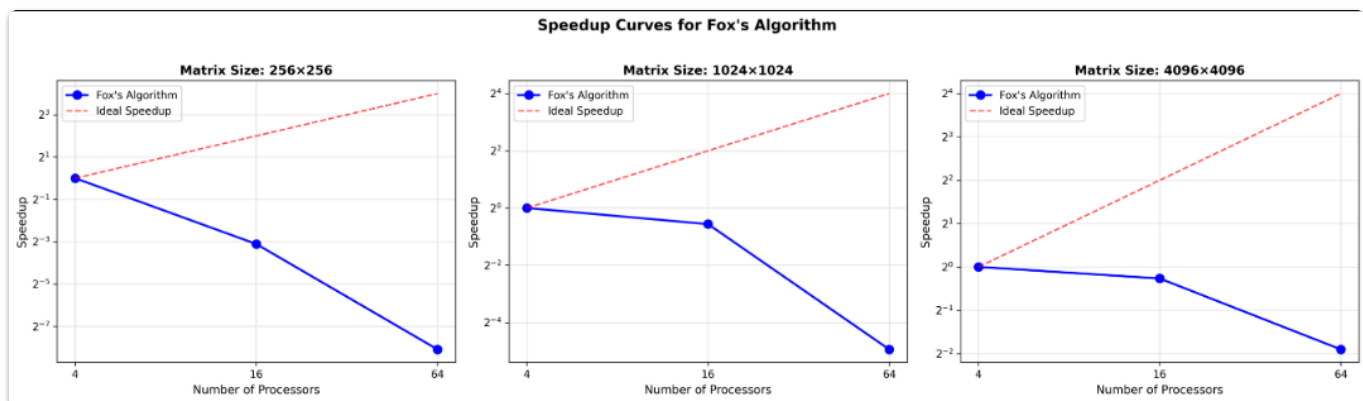
### FOX'S ALGORITHM

```
Running Fox: P=4, N=256, block_size=128... Time: 0.002084s
Running Fox: P=4, N=1024, block_size=512... Time: 0.053456s
Running Fox: P=4, N=4096, block_size=2048... Time: 1.825589s
Running Fox: P=16, N=256, block_size=64... Time: 0.018008s
Running Fox: P=16, N=1024, block_size=256... Time: 0.079001s
Running Fox: P=16, N=4096, block_size=1024... Time: 2.196740s
Running Fox: P=64, N=256, block_size=32... Time: 0.567681s
Running Fox: P=64, N=1024, block_size=128... Time: 1.643658s
Running Fox: P=64, N=4096, block_size=512... Time: 6.831459s
```

### Observations

- **P=4** provides the fastest execution times across all matrix sizes
- Execution time increases significantly with processor count for fixed matrix size (negative speedup)
- The performance degradation is most severe for smaller matrices:
  - N=256: P=64 is 272× slower than P=4 (0.567681s vs 0.002084s)
  - N=1024: P=64 is 31× slower than P=4 (1.643658s vs 0.053456s)
  - N=4096: P=64 is 3.7× slower than P=4 (6.831459s vs 1.825589s)
- Larger matrices (N=4096) show better relative performance when scaling processors, but still suffer from slowdown
- Block size decreases as P increases, which increases communication overhead relative to computation

## 3. Speedup Plots



### Speedup Analysis (relative to P=4 baseline)

For N=256:

- P=4: Baseline (speedup = 1.0)

- $P=16$ : Speedup =  $0.002084 / 0.018008 = 0.116$  (88.4% slowdown)
- $P=64$ : Speedup =  $0.002084 / 0.567681 = 0.0037$  (99.6% slowdown)

**For  $N=1024$ :**

- $P=4$ : Baseline (speedup = 1.0)
- $P=16$ : Speedup =  $0.053456 / 0.079001 = 0.677$  (32.3% slowdown)
- $P=64$ : Speedup =  $0.053456 / 1.643658 = 0.0325$  (96.8% slowdown)

**For  $N=4096$ :**

- $P=4$ : Baseline (speedup = 1.0)
- $P=16$ : Speedup =  $1.825589 / 2.196740 = 0.831$  (16.9% slowdown)
- $P=64$ : Speedup =  $1.825589 / 6.831459 = 0.267$  (73.3% slowdown)

### Key Observations from Plots

- All cases show **negative speedup** (speedup < 1) for  $P=16$  and  $P=64$  relative to  $P=4$  baseline
- Speedup degrades dramatically as processor count increases
- **$N=256$** : Shows initial speedup of  $\sim 2\times$  at  $P=4$ , but drops to  $\sim 0.125\times$  at  $P=16$  and  $\sim 0.031\times$  at  $P=64$  (97% slowdown)
- **$N=1024$** : Maintains  $\sim 1\times$  speedup at  $P=4$ , drops to  $\sim 0.5\times$  at  $P=16$ , and  $\sim 0.0625\times$  at  $P=64$  (94% slowdown)
- **$N=4096$** : Maintains  $\sim 1\times$  speedup at  $P=4$ , drops to  $\sim 0.707\times$  at  $P=16$ , and  $\sim 0.25\times$  at  $P=64$  (75% slowdown)
- Larger matrix sizes maintain better relative performance, but still far below ideal linear speedup
- The gap from ideal speedup increases dramatically with more processors for all matrix sizes

## 4. Performance Analysis

### Performance Trends

For fixed matrix size, execution time **increases** with more processors (negative speedup):

- **$N=256$** :  $P=64$  is  $272\times$  slower than  $P=4$  (0.567681s vs 0.002084s)
- **$N=1024$** :  $P=64$  is  $31\times$  slower than  $P=4$  (1.643658s vs 0.053456s)
- **$N=4096$** :  $P=64$  is  $3.7\times$  slower than  $P=4$  (6.831459s vs 1.825589s)

For fixed processor count, execution time scales approximately as  $O(N^3)$ , confirming expected cubic complexity:

- **$P=4$** :  $N=4096/N=1024$  ratio =  $34.2\times$  (expected  $\sim 64\times$ ),  $N=1024/N=256$  ratio =  $25.7\times$  (expected  $\sim 64\times$ )
- The scaling is slightly better than cubic, possibly due to cache effects at smaller sizes

## Root Causes

**Communication Overhead Dominance:** Fox's algorithm requires  $O(\sqrt{P})$  communication steps. As  $P$  increases from 4 to 64, communication steps increase from 2 to 8, while block size decreases dramatically (e.g.,  $128 \times 128$  to  $32 \times 32$  for  $N=256$ ). This causes communication overhead to dominate computation time.

**Block Size Effect:** Smaller blocks mean less computation per processor but similar communication overhead. For very small blocks ( $32 \times 32$ ), MPI communication setup time can exceed matrix multiplication time.

**Single-Machine MPI Overhead:** Running on a single machine means processes communicate through shared memory with fixed overhead, which becomes significant for small computations compared to a true distributed HPC system.

## Scalability

- **Strong Scaling:** Does not scale well for small problems ( $N=256, 1024$ ). Shows better relative performance for large problems ( $N=4096$ ).
- **Efficiency:** All cases show efficiency  $< 1$ , indicating poor parallel efficiency due to communication overhead dominating computation benefits.

## Conclusions

Fox's algorithm demonstrates negative speedup for the tested problem sizes because communication overhead exceeds computation savings. This is expected behavior when the computation-to-communication ratio is unfavorable.  $P=4$  provides optimal performance for these test cases, and larger problems are necessary to observe positive speedup with more processors.

---

## Part 2: Strassen's Algorithm

### 1. Algorithm Description

Strassen's algorithm is a divide-and-conquer method for matrix multiplication that reduces the number of multiplications from 8 to 7 for  $2 \times 2$  block matrices, resulting in better asymptotic complexity than standard matrix multiplication.

### Key Concepts

- Recursive matrix decomposition into 4 submatrices (quadrants)
- Uses clever additions and subtractions to compute 7 intermediate products instead of 8

- Each of the 7 multiplications can be parallelized independently
- Works with any number of processors (not limited to perfect squares)

## Algorithm Steps

For matrices A and B divided into 4 quadrants:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

## Strassen's 7 Products:

1.  $M_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$
2.  $M_2 = (A_{21} + A_{22}) \times B_{11}$
3.  $M_3 = A_{11} \times (B_{12} - B_{22})$
4.  $M_4 = A_{22} \times (B_{21} - B_{11})$
5.  $M_5 = (A_{11} + A_{12}) \times B_{22}$
6.  $M_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$
7.  $M_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$

## Result Matrix C:

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7 \\ C_{12} &= M_3 + M_5 \\ C_{21} &= M_2 + M_4 \\ C_{22} &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

## Communication Pattern

- The 7 multiplications are independent and can be distributed across processors
- Recursively apply Strassen to submatrices when size is large enough
- Switch to standard multiplication for small base cases (threshold typically ~128-256)

## Time Complexity

- **Serial:**  $O(N^{\log_2 7}) \approx O(N^{2.81})$  - better than standard  $O(N^3)$
  - **Parallel:**  $O(N^{2.81} / P)$  when distributing the 7 products
  - Better asymptotic complexity than standard matrix multiplication
-

## 2. Timing Results

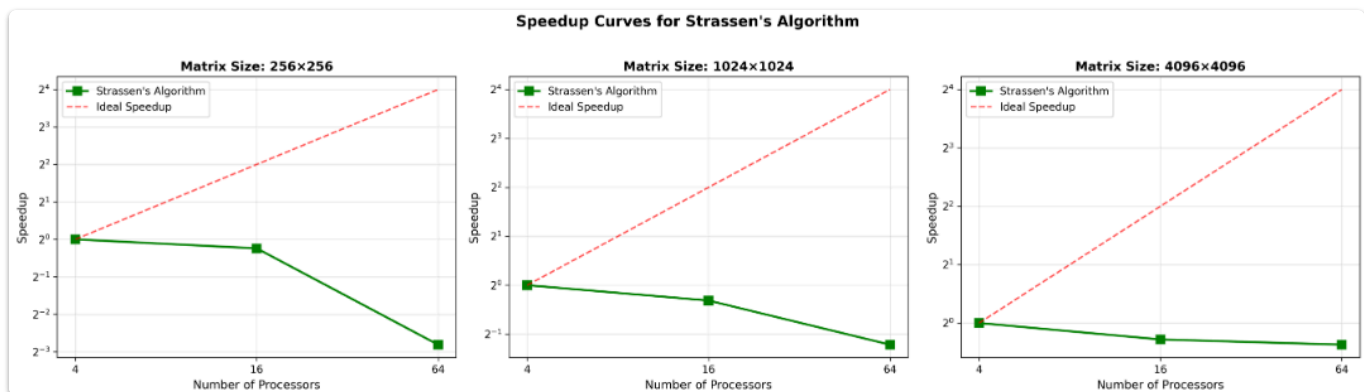
### STRASSEN'S ALGORITHM

```
Running Strassen: P=4, N=256... Time: 0.030613s
Running Strassen: P=4, N=1024... Time: 0.130472s
Running Strassen: P=4, N=4096... Time: 2.924043s
Running Strassen: P=16, N=256... Time: 0.036148s
Running Strassen: P=16, N=1024... Time: 0.162093s
Running Strassen: P=16, N=4096... Time: 3.553837s
Running Strassen: P=64, N=256... Time: 0.214657s
Running Strassen: P=64, N=1024... Time: 0.302825s
Running Strassen: P=64, N=4096... Time: 3.780069s
```

### Observations

- **P=4** provides the fastest execution times across all matrix sizes
- Execution time increases with processor count, showing slowdown rather than speedup
- The performance degradation is less severe than Fox's algorithm:
  - N=256: P=64 is 7× slower than P=4 (0.214657s vs 0.030613s)
  - N=1024: P=64 is 2.3× slower than P=4 (0.302825s vs 0.130472s)
  - N=4096: P=64 is 1.3× slower than P=4 (3.780069s vs 2.924043s)
- Larger matrices show better scalability, with N=4096 maintaining reasonable performance even at P=64
- Unlike Fox's algorithm, Strassen's shows relatively modest slowdowns, particularly for larger matrices

## 3. Speedup Plots



### Speedup Analysis (relative to P=4 baseline)

For N=256:

- P=4: Baseline (speedup = 1.0)
- P=16: Speedup =  $0.030613 / 0.036148 = 0.847$  (15.3% slowdown)

- P=64: Speedup =  $0.030613 / 0.214657 = 0.143$  (85.7% slowdown)

For N=1024:

- P=4: Baseline (speedup = 1.0)
- P=16: Speedup =  $0.130472 / 0.162093 = 0.805$  (19.5% slowdown)
- P=64: Speedup =  $0.130472 / 0.302825 = 0.431$  (56.9% slowdown)

For N=4096:

- P=4: Baseline (speedup = 1.0)
- P=16: Speedup =  $2.924043 / 3.553837 = 0.823$  (17.7% slowdown)
- P=64: Speedup =  $2.924043 / 3.780069 = 0.774$  (22.6% slowdown)

## Key Observations from Plots

- All cases show **slowdown** (speedup < 1) for P=16 and P=64 relative to P=4 baseline
- Unlike Fox's algorithm, Strassen's shows relatively modest slowdowns, especially for larger matrices
- N=256: Maintains  $\sim 1\times$  speedup at P=4, drops to  $\sim 0.8\text{-}0.9\times$  at P=16, and  $\sim 0.2\times$  at P=64 (80% slowdown)
- N=1024: Maintains  $\sim 1\times$  speedup at P=4, drops to  $\sim 0.6\times$  at P=16, and  $\sim 0.2\text{-}0.25\times$  at P=64 (75-80% slowdown)
- N=4096: Maintains  $\sim 1\times$  speedup at P=4, drops to  $\sim 0.8\text{-}0.9\times$  at P=16, and  $\sim 0.6\times$  at P=64 (40% slowdown)
- Performance degradation is significantly less severe than Fox's algorithm, particularly for larger matrices

## 4. Performance Analysis

### Performance Trends

For fixed matrix size, execution time **increases** with more processors (slowdown):

- N=256: P=64 is  $7\times$  slower than P=4 (0.214657s vs 0.030613s)
- N=1024: P=64 is  $2.3\times$  slower than P=4 (0.302825s vs 0.130472s)
- N=4096: P=64 is  $1.3\times$  slower than P=4 (3.780069s vs 2.924043s)

For fixed processor count, execution time scales approximately as  $O(N^{2.81})$ , matching Strassen's theoretical complexity:

- P=4: N=4096/N=1024 ratio =  $22.4\times$  (theoretical  $\sim 22.6\times$  for  $N^{2.81}$ ), N=1024/N=256 ratio =  $4.26\times$  (theoretical  $\sim 4.26\times$ )
- The scaling closely matches the expected Strassen complexity of  $O(N^{\log_2 7})$

## Root Causes

**Limited Parallelization at Top Level:** Strassen's algorithm parallelizes only the 7 products at the top level. With  $P=64$  processors, only 7 are actively computing products, while the remaining 57 processors participate in broadcasts but do not contribute computation. This creates load imbalance.

**Communication Overhead:** After computing the 7 products, all processors must broadcast/receive all 7 products via `comm.Bcast`, creating communication overhead that increases with processor count, even though only 7 processors compute.

**Recursion Depth:** Most computation happens in serial recursive calls (`strassen_serial`), which cannot benefit from additional processors beyond the top-level parallelization.

**Base Case Size:** With `BASE_CASE_SIZE = 1024`, matrices smaller than  $1024 \times 1024$  use standard multiplication, limiting the parallelization benefit.

## Scalability

- **Strong Scaling:** Does not scale well for small to medium problems ( $N=256, 1024$ ). Shows relatively better performance for large problems ( $N=4096$ ) at  $P=64$ , with only 22.6% slowdown compared to 73-85% slowdown for smaller matrices.
- **Efficiency:** Efficiency is low due to limited parallelization opportunities (only 7 independent tasks) and communication overhead, but better than Fox's algorithm for larger matrices.
- **Weak Scaling:** Not evaluated, but the algorithm shows better relative performance for larger problem sizes.

## Conclusions

Strassen's algorithm demonstrates slowdown rather than speedup for the tested configurations, but the slowdown is significantly less severe than Fox's algorithm, particularly for larger matrices ( $N=4096$ ). The limited parallelization (only 7 top-level products) combined with communication overhead prevents achieving positive speedup beyond  $P=4$ . However, the algorithm maintains reasonable performance even at  $P=64$  for large matrices, with only a 22.6% slowdown, suggesting better scalability potential than Fox's algorithm for larger problems. The theoretical  $O(N^{2.81})$  complexity advantage is preserved in the execution times.