HO CHI MINH CITY, UNIVERSITY OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEER

Assignment Report Group 4 CO3067

Parallel Computing

Semester: 251

Students: Théo Bloch - 2460078
Nguyen Nhu Tinh Anh - 2252034
Nguyen Phuc Thanh Danh - 2252102

HO CHI MINH CITY

Contents

1 Introduction

This document provides comprehensive documentation for parallel matrix multiplication implementations using different parallel programming paradigms: MPI (Message Passing Interface), OpenMP (Open Multi-Processing), and hybrid MPI+OpenMP approaches. The project implements both naive (standard) and Strassen's algorithm for matrix multiplication.

1.1 Project Objectives

- Implement parallel matrix multiplication using MPI for distributed memory systems
- Implement parallel matrix multiplication using OpenMP for shared memory systems
- Implement hybrid MPI+OpenMP approach combining both paradigms
- Compare naive and Strassen's algorithm performance
- Benchmark and analyze scalability across different problem sizes

1.2 Algorithms Implemented

1. **Naive Matrix Multiplication**: Standard $O(n^3)$ algorithm
2. **Strassen's Algorithm**: Divide-and-conquer approach with $O(n^{2.807})$ complexity

2 MPI Implementation

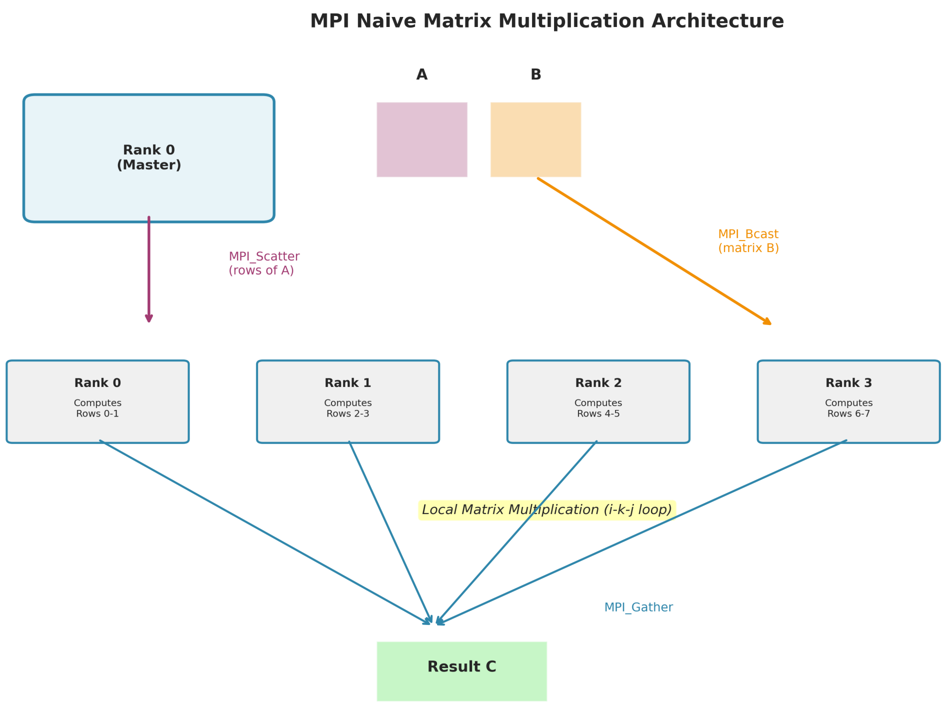

Figure 1: MPI Naive Matrix Multiplication Architecture

The Message Passing Interface (MPI) implementations leverage distributed memory parallelism, allowing matrix multiplication to scale across multiple nodes in a cluster. We implemented two variants: a straightforward naive algorithm using standard triple-nested loops, and Strassen's divide-and-conquer algorithm adapted for distributed execution.

2.1 MPI Naive Matrix Multiplication

The naive implementation employs a row-wise decomposition strategy where matrix A is partitioned horizontally across processes while matrix B is broadcast to all processes. This approach minimizes communication complexity while maintaining load balance. Each process computes a subset of output rows with computational complexity $O(n^3/p)$ where p is the number of processes.

Matrix Distribution: Process i receives rows $[i \times \frac{N}{p}, (i+1) \times \frac{N}{p})$ of matrix A. The entire matrix B is broadcast to all processes using `MPI_Bcast`, enabling each process to independently compute its assigned output rows.

Key Implementation Features:

- **Cache Optimization**: i-k-j loop ordering for better cache locality
- **Load Balancing**: Equal row distribution ensures uniform workload
- **Communication Pattern**: One scatter operation for A, one broadcast for B, one gather for C
- **Verification**: Optional serial verification on rank 0 for correctness checking

The communication overhead is dominated by the broadcast of matrix B ($O(n^2)$ elements) and gathering results ($O(n^2/p)$ per process), which becomes negligible for large matrices where computation is $O(n^3/p)$.

File: mpi-naive/mpi-naive.h

```
#ifndef MPI_NAIVE_H
#define MPI_NAIVE_H

#include <mpi.h>
#include <iostream>
#include <ctime>
#include <vector>
#include <cstdlib>
#include <cmath>
#include <iomanip>

void initializeMatrices(int N, int rank,
                       std::vector<int>& A,
                       std::vector<int>& B,
                       std::vector<int>& C);

void distributeMatrices(int N, int rank,
                       const std::vector<int>& A,
                       std::vector<int>& local_a,
                       std::vector<int>& B,
                       int rows_per_proc);

void localMatrixComputation(int N, int rows_per_proc,
                            const std::vector<int>& local_a,
                            const std::vector<int>& B,
                            std::vector<int>& local_c,
                            double& local_time);

void gatherResults(int N, int rank, int rows_per_proc,
                   const std::vector<int>& local_c,
                   std::vector<int>& C);

double computeMaxLocalTime(double local_time, int rank);

void serialVerify(int N, const std::vector<int>& A,
                  const std::vector<int>& B,
                  std::vector<int>& C_verify);

bool verifyResults(int N, const std::vector<int>& C,
                   const std::vector<int>& C_verify,
                   int rank);

#endif
```

Key Functions:

- `initializeMatrices()`: Initialize random matrices on rank 0
- `distributeMatrices()`: Use MPI_Scatter and MPI_Bcast to distribute data
- `localMatrixComputation()`: Each process computes its portion using cache-optimized loop ordering (i-k-j)
- `gatherResults()`: Collect results using MPI_Gather

Matrix Distribution Strategy:

$$\text{rows\_per\_process} = \frac{N}{p} \tag{1}$$

Process i receives rows $[i \times \text{rows\_per\_process}, (i+1) \times \text{rows\_per\_process})$
Communication Pattern:

1. **Scatter**: Distribute rows of matrix A to all processes
2. **Broadcast**: Send entire matrix B to all processes
3. **Computation**: Local matrix multiplication
4. **Gather**: Collect results back to rank 0

2.2 MPI Strassen Matrix Multiplication

Algorithm Overview:

Strassen's algorithm reduces matrix multiplication to 7 recursive multiplications instead of 8, achieving better asymptotic complexity.

Complexity: $O(n^{\log_2 7}) \approx O(n^{2.807})$

The Seven Products:

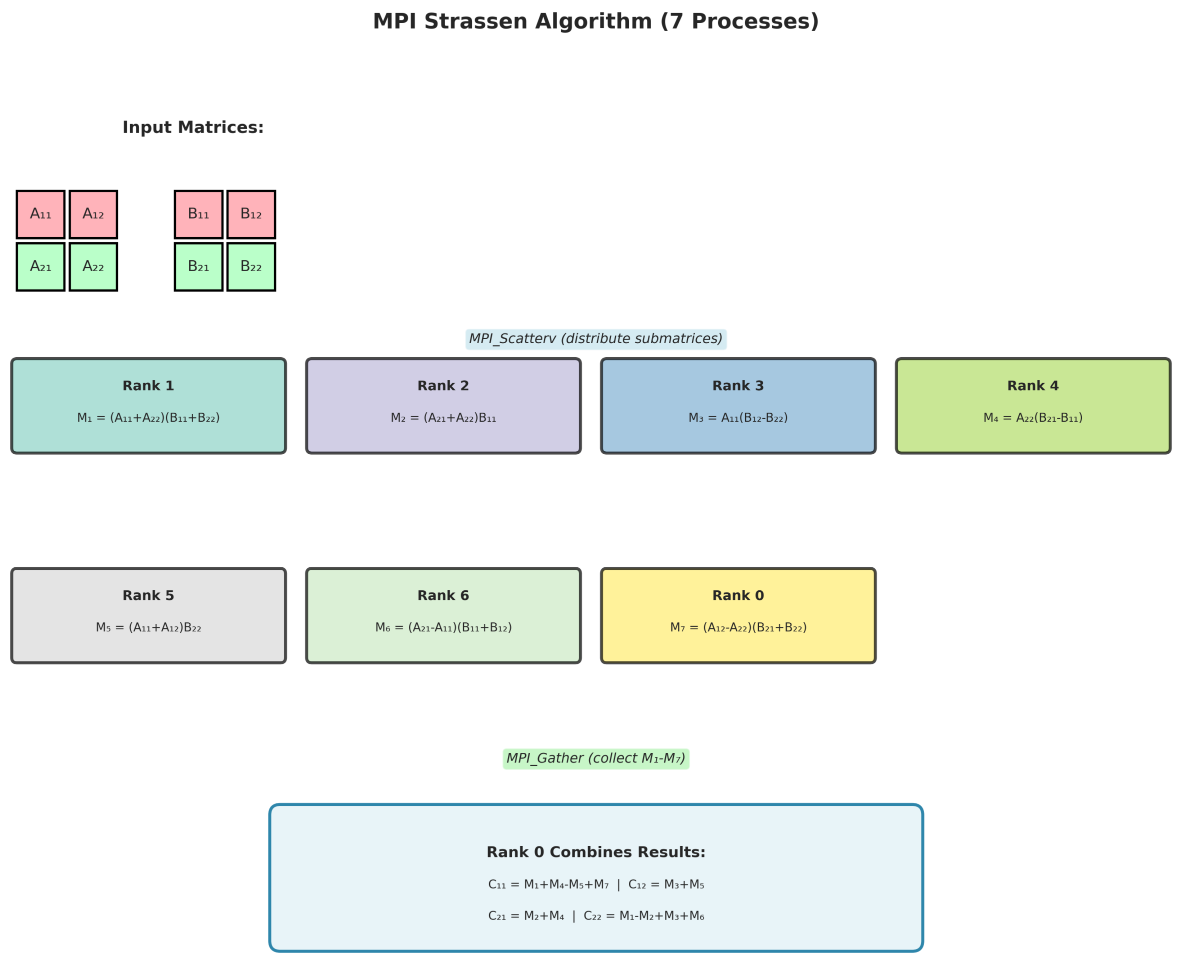
MPI Strassen Algorithm (7 Processes)
Input Matrices:
A_{11} A_{12} A_{21} A_{22}
B_{11} B_{12} B_{21} B_{22}
MPI_Scatterv (distribute submatrices)
Rank 1
$M_1 = (A_{11}+A_{22})(B_{11}+B_{22})$
Rank 2
$M_2 = (A_{21}+A_{22})B_{11}$
Rank 3
$M_3 = A_{11}(B_{12}-B_{22})$
Rank 4
$M_4 = A_{22}(B_{21}-B_{11})$
Rank 5
$M_5 = (A_{11}+A_{12})B_{22}$
Rank 6
$M_6 = (A_{21}-A_{11})(B_{11}+B_{12})$
Rank 0
$M_7 = (A_{12}-A_{22})(B_{21}+B_{22})$
MPI_Gather (collect M_1-M_7)
Rank 0 Combines Results:
$C_{11} = M_1+M_4-M_5+M_7$ | $C_{12} = M_3+M_5$
$C_{21} = M_2+M_4$ | $C_{22} = M_1-M_2+M_3+M_6$

Figure 2: MPI Strassen Algorithm - 7 Process Architecture

For matrices partitioned into blocks:

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

The seven products are:

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22}) \tag{2}$$
$$M_2 = (A_{21} + A_{22})B_{11} \tag{3}$$
$$M_3 = A_{11}(B_{12} - B_{22}) \tag{4}$$
$$M_4 = A_{22}(B_{21} - B_{11}) \tag{5}$$
$$M_5 = (A_{11} + A_{12})B_{22} \tag{6}$$
$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12}) \tag{7}$$
$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22}) \tag{8}$$

Result matrix blocks:

$$C_{11} = M_1 + M_4 - M_5 + M_7 \tag{9}$$
$$C_{12} = M_3 + M_5 \tag{10}$$
$$C_{21} = M_2 + M_4 \tag{11}$$
$$C_{22} = M_1 - M_2 + M_3 + M_6 \tag{12}$$

MPI Parallelization Strategy:
The implementation uses exactly 7 processes, one for each Strassen product:

- **Process 0**: Coordinates and computes M_7
- **Process 1**: Computes M_1
- **Process 2**: Computes M_2
- **Process 3**: Computes M_3
- **Process 4**: Computes M_4
- **Process 5**: Computes M_5
- **Process 6**: Computes M_6

Implementation Structure:
File: mpi-strassen/mpi-strassen.h

```
#ifndef MPI_STRASSEN_H
#define MPI_STRASSEN_H

#include <mpi.h>
#include <iostream>
#include <cmath>
#include <chrono>
#include <vector>
#include <random>
#include <cstring>

#define LOWER_B 0.0
```

```
#define UPPER_B 1.0
#define THRESHOLD 128

class Timer {
    std::chrono::high_resolution_clock::time_point start_;
public:
    void start() {
        start_ = std::chrono::high_resolution_clock::now();
    }
    float elapse() {
        auto end = std::chrono::high_resolution_clock::now();
        return std::chrono::duration<float>(end - start_).count();
    }
};

std::vector<float> createRandomMatrix(int size, int seed);

void naiveMultiply(int n, const float *A, int lda,
                   const float *B, int ldb,
                   float *C, int ldc);

void addMatrix(int n, const float *A, int lda,
               const float *B, int ldb,
               float *C, int ldc);

void subtractMatrix(int n, const float *A, int lda,
                    const float *B, int ldb,
                    float *C, int ldc);

void strassenSerial(int n, const float *A, int lda,
                    const float *B, int ldb,
                    float *C, int ldc,
                    float *work);

void strassen_mpi_wrapper(int N, int rank, int numProcs,
                          int *sendcounts, int *displs,
                          const float *A, int lda,
                          const float *B, int ldb,
                          float *C, int ldc);

#endif
```

3 OpenMP Implementation

The OpenMP implementations leverage shared-memory parallelism using thread-based task decomposition. OpenMP provides a simpler programming model compared to MPI, as all threads share the same address space. We implemented both naive and Strassen algorithms using OpenMP's task-based parallelism combined with recursive divide-and-conquer strategies for optimal load balancing.

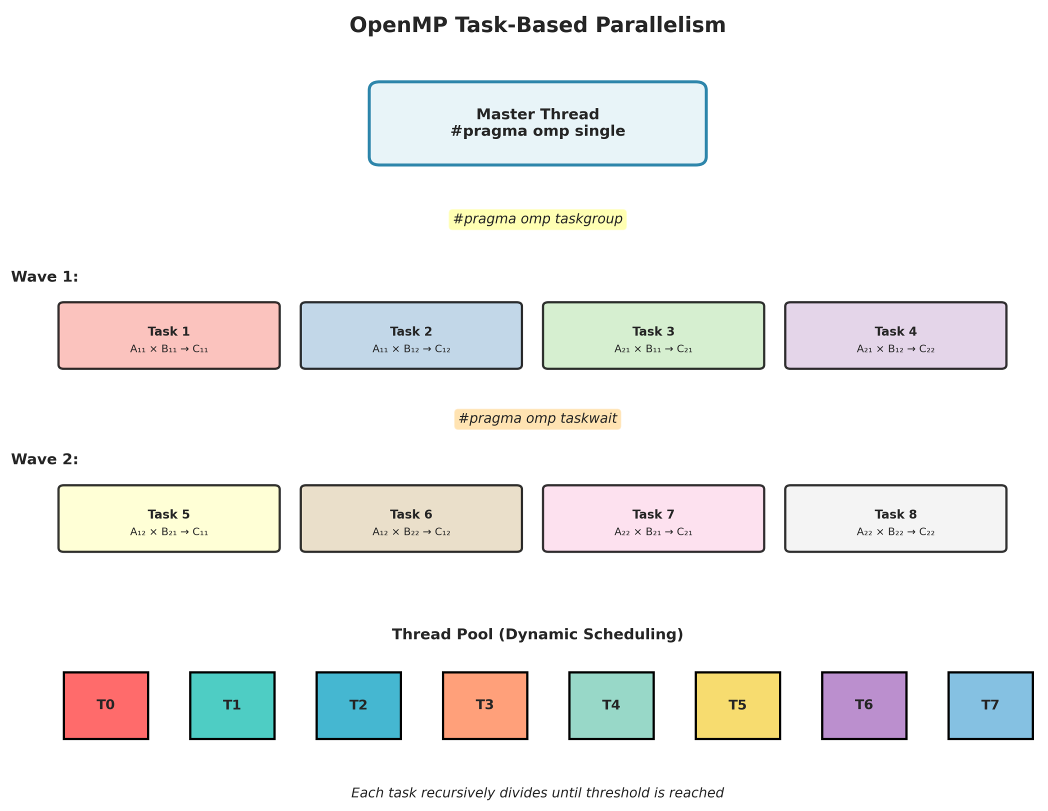

Figure 3: OpenMP Task-Based Parallelism Architecture

3.1 OpenMP Naive with Divide-and-Conquer

The OpenMP naive implementation combines traditional matrix multiplication with a divide-and-conquer decomposition strategy. Unlike the straightforward triple-nested loop approach, this implementation recursively subdivides matrices into quadrants and uses OpenMP tasks to parallelize independent computations. This approach provides better load balancing and cache locality compared to simple loop parallelization.

Algorithm Description:

The OpenMP naive implementation uses a recursive divide-and-conquer approach with task-based parallelism. This allows efficient load balancing across threads.

Recursive Decomposition:

The matrix multiplication $C = AB$ is decomposed as:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

This results in 8 smaller multiplications:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \tag{13}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22} \tag{14}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \tag{15}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22} \tag{16}$$

OpenMP Task Parallelism:

```
void recursiveMatMul(int n, const float *A, int lda,
                     const float *B, int ldb,
                     float *C, int ldc, int threshold) {
    if (n <= threshold || n % 2 != 0) {
        naiveAddMultiply(n, A, lda, B, ldb, C, ldc);
        return;
    }

    int m = n / 2;

    #pragma omp taskgroup
    {
        // First wave of 4 tasks
        #pragma omp task
        recursiveMatMul(m, A11, lda, B11, ldb, C11, ldc,
   threshold);

        #pragma omp task
        recursiveMatMul(m, A11, lda, B12, ldb, C12, ldc,
   threshold);

        #pragma omp task
        recursiveMatMul(m, A21, lda, B11, ldb, C21, ldc,
   threshold);

        #pragma omp task
        recursiveMatMul(m, A21, lda, B12, ldb, C22, ldc,
   threshold);

        #pragma omp taskwait

        // Second wave of 4 tasks
        #pragma omp task
        recursiveMatMul(m, A12, lda, B21, ldb, C11, ldc,
   threshold);
        // ... remaining tasks
    }
}
```

3.2 OpenMP Strassen Implementation

The OpenMP Strassen implementation adapts the seven-product algorithm for shared-memory parallelism. Each recursive call potentially spawns OpenMP tasks for the seven products, allowing the runtime to dynamically schedule work across available threads. This implementation includes sophisticated optimizations such as adaptive thresholding and cache-aware base cases to maximize performance across different matrix sizes and core counts.

Parallel Strategy:

The OpenMP Strassen implementation uses:

- Task-based parallelism for the 7 Strassen products
- Depth-limited recursion to control task creation overhead
- SIMD vectorization for base case computations

Implementation Highlights:
File: openmp-strassen/openmap-strassen.h

```
void strassenParallel(int n, const float *A, int lda,
                      const float *B, int ldb,
                      float *C, int ldc,
                      int depth, int max_depth, int threshold) {
    if (depth >= max_depth || n % 2 != 0) {
        if (n % 2 == 0) {
            size_t stackSize = (size_t)(3 * n * n);
            std::vector<float> serialStack(stackSize);
            strassenSerial(n, A, lda, B, ldb, C, ldc,
                           serialStack.data(), threshold);
        } else {
            naiveMultiply(n, A, lda, B, ldb, C, ldc);
        }
        return;
    }

    int m = n / 2;
    std::vector<float> results(7 * m * m);

    #pragma omp taskgroup
    {
        // Create 7 tasks for M1-M7
        #pragma omp task shared(results)
        {
            // Compute M2 = (A21 + A22)B11
            std::vector<float> T(m * m);
            addMatrix(m, A21, lda, A22, lda, T.data(), m);
            strassenParallel(m, T.data(), m, B11, ldb,
                             M2, m, depth + 1, max_depth, threshold);
        }
        // ... remaining 6 tasks
    }

    // Combine results
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < m; j++) {
            int k = i * m + j;
            C11[k] = M1[k] + M4[k] - M5[k] + M7[k];
            C12[k] = M3[k] + M5[k];
            C21[k] = M2[k] + M4[k];
            C22[k] = M1[k] - M2[k] + M3[k] + M6[k];
        }
    }
}
```

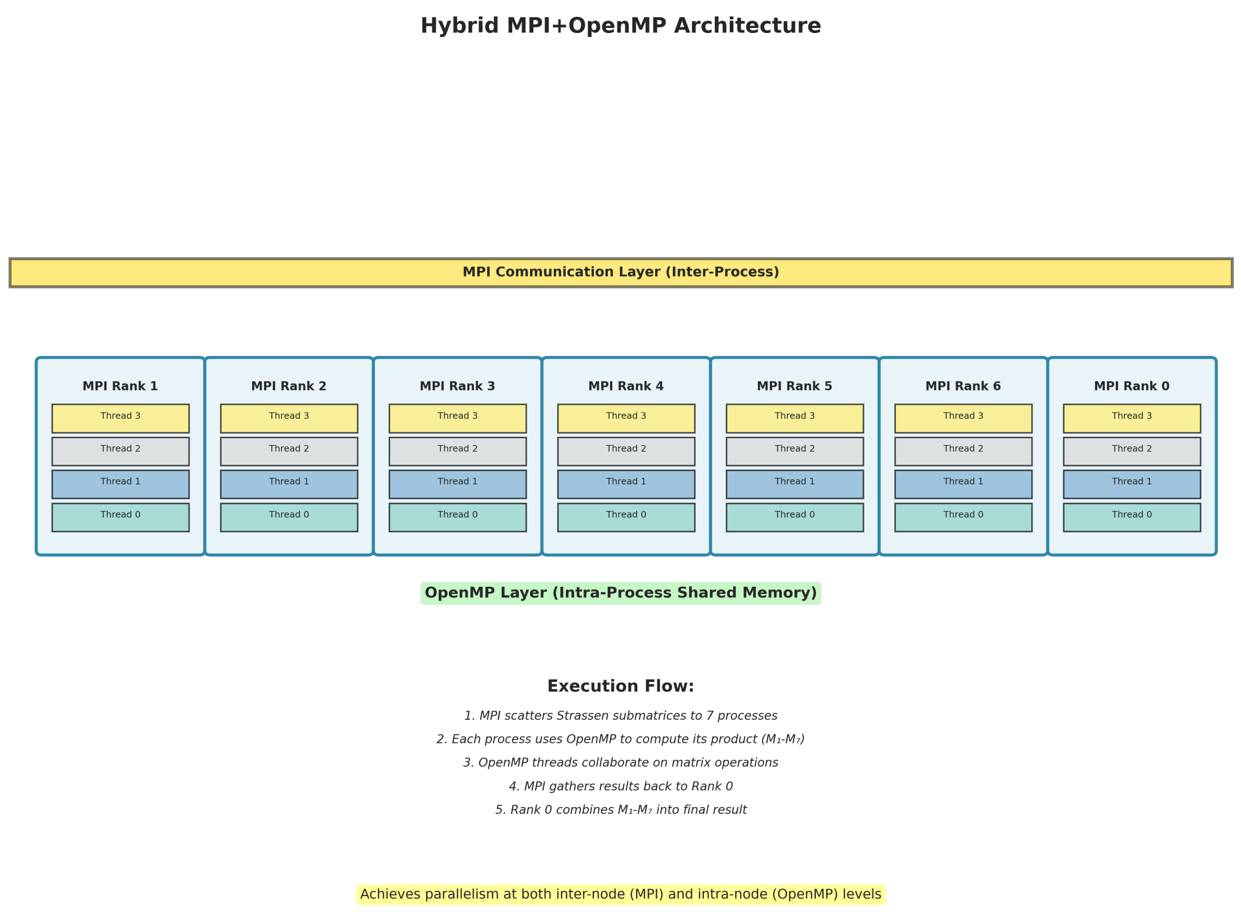
Hybrid MPI+OpenMP Architecture
MPI Communication Layer (Inter-Process)
MPI Rank 1
MPI Rank 2
MPI Rank 3
MPI Rank 4
MPI Rank 5
MPI Rank 6
MPI Rank 0
Thread 3
Thread 2
Thread 1
Thread 0
OpenMP Layer (Intra-Process Shared Memory)
Execution Flow:
1. MPI scatters Strassen submatrices to 7 processes
2. Each process uses OpenMP to compute its product (M₁-M₇)
3. OpenMP threads collaborate on matrix operations
4. MPI gathers results back to Rank 0
5. Rank 0 combines M₁-M₇ into final result
Achieves parallelism at both inter-node (MPI) and intra-node (OpenMP) levels

Figure 4: Hybrid MPI+OpenMP Architecture - Multi-Level Parallelism

4 Hybrid MPI+OpenMP Implementation

4.1 Architecture Overview

The hybrid implementation represents the most sophisticated parallelization strategy, combining distributed-memory MPI for inter-node communication with shared-memory OpenMP for intra-node parallelism. This two-level approach is particularly effective on modern HPC clusters where each node contains multiple cores. The implementation uses 7 MPI processes (matching Strassen's requirement), with each process spawning multiple OpenMP threads to fully utilize available hardware resources.

Key architectural components:

- **MPI Layer**: Distributes the seven Strassen products across processes, handles inter-node data movement
- **OpenMP Layer**: Each MPI process uses OpenMP threads to parallelize its assigned matrix product computation
- **Load Balancing**: MPI provides coarse-grained parallelism (7-way), OpenMP provides fine-grained parallelism within each process
- **Memory Efficiency**: Shared memory within nodes reduces communication overhead compared to pure MPI

Communication Strategy:

The communication pattern carefully minimizes data movement while ensuring all processes have the necessary submatrices:

1. **Matrix Distribution**: Rank 0 packs submatrices and scatters to 7 processes
2. **Local Computation**: Each process uses OpenMP to compute its Strassen product
3. **Result Collection**: MPI_Gather collects results to rank 0
4. **Final Combination**: Rank 0 combines the 7 products into final result

4.2 Thread Safety

The implementation ensures thread safety through:

- Thread-local storage for temporary matrices
- Task-based parallelism avoiding race conditions
- Proper synchronization using `#pragma omp taskwait`

5 Build System

All implementations include comprehensive Makefiles for easy compilation and testing.

5.1 Makefile Structure

Each implementation includes a comprehensive Makefile with:

- Optimized compilation flags: `-O3 -march=native`
- Automated testing targets for different matrix sizes
- Benchmark automation with multiple runs
- Result collection and summary generation

Example Compilation Flags:

```
# MPI Naive
CXX = mpicxx
CXXFLAGS = -std=c++11 -O3 -Wall -Wextra -march=native

# OpenMP
CXX = g++
CXXFLAGS = -std=c++11 -O3 -fopenmp -Wall -Wextra -march=native

# Hybrid
CXX = mpicxx
CXXFLAGS = -std=c++11 -O3 -fopenmp -Wall -Wextra -march=native
LDFLAGS = -fopenmp
```

6 Experimental Results

This section presents comprehensive benchmark results from testing all implementations across multiple computing environments.

6.1 Test Environment

HPCC Cluster (MPI Tests):

- **Node**: MPI-node5
- **CPU Cores**: 8 cores per node
- **MPI Version**: MPICH 4.0
- **Network**: 10.1.8.0/24
- **Date**: December 12, 2025

OpenMP Test Machines:
Machine 1:

- High-performance workstation
- Multiple CPU cores (up to 16 threads)
- Best performance observed

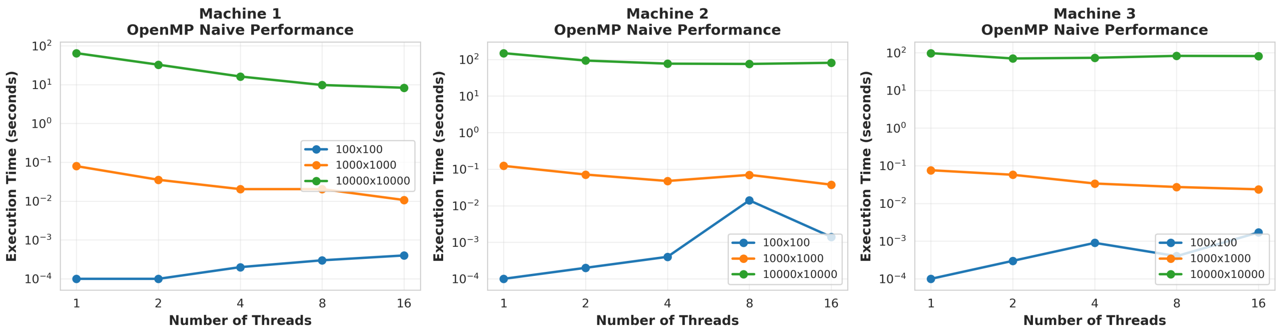

Figure 5: OpenMP Naive Performance Comparison Across Three Machines

Table 1: OpenMP Naive Performance - Machine 1 (Complete Results)

Matrix Size	Threads	Time (s)	Speedup
100×100	1	0.0001	1.00×
100×100	2	0.0001	1.00×
100×100	4	0.0002	0.50×
100×100	8	0.0003	0.33×
100×100	16	0.0004	0.25×
1000×1000	1	0.0797	1.00×
1000×1000	2	0.0355	2.24×
1000×1000	4	0.0204	3.91×
1000×1000	8	0.0204	3.91×
1000×1000	16	0.0107	7.45×
10000×10000	1	64.6434	1.00×
10000×10000	2	32.6183	1.98×
10000×10000	4	16.1950	3.99×
10000×10000	8	9.7749	6.61×
10000×10000	16	8.3111	7.78×

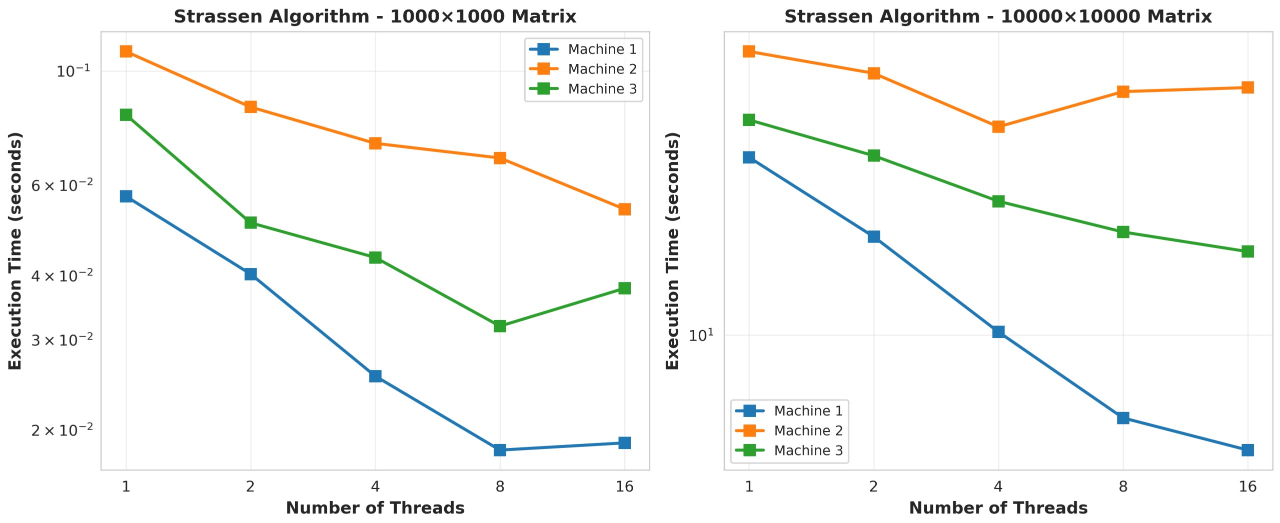

Figure 6: OpenMP Strassen Performance Across Three Machines

Table 2: OpenMP Naive Performance - Machine 2 (Complete Results)

Matrix Size	**Threads**	**Time (s)**	**Speedup**
100×100	1	0.0001	1.00×
100×100	2	0.0002	0.50×
100×100	4	0.0004	0.25×
100×100	8	0.0140	0.01×
100×100	16	0.0014	0.07×
1000×1000	1	0.1230	1.00×
1000×1000	2	0.0712	1.73×
1000×1000	4	0.0474	2.59×
1000×1000	8	0.0698	1.76×
1000×1000	16	0.0377	3.26×
10000×10000	1	149.0415	1.00×
10000×10000	2	93.5300	1.59×
10000×10000	4	76.8988	1.94×
10000×10000	8	75.5792	1.97×
10000×10000	16	81.1862	1.84×

Table 3: OpenMP Strassen Performance - Complete Results (1000×1000)

Machine	**Threads**	**Time (s)**	**Speedup**	**Efficiency (%)**
	1	0.0571	1.00×	100.0
	2	0.0403	1.42×	70.9
Machine 1	4	0.0255	2.24×	56.0
	8	0.0183	3.12×	39.0
	16	0.0256	2.23×	14.0
	1	0.1093	1.00×	100.0
	2	0.0852	1.28×	64.2
Machine 2	4	0.0724	1.51×	37.7
	8	0.0678	1.61×	20.1
	16	0.0539	2.03×	12.7
	1	0.0823	1.00×	100.0
	2	0.0507	1.62×	81.2
Machine 3	4	0.0434	1.90×	47.4
	8	0.0319	2.58×	32.3
	16	0.0378	2.18×	13.6

Table 4: OpenMP Strassen Performance - Complete Results (10000×10000)

Machine	Threads	Time (s)	Speedup	Efficiency (%)
Machine 1	1	32.9442	1.00×	100.0
	2	19.3577	1.70×	85.1
	4	10.2281	3.22×	80.5
	8	5.7473	5.73×	71.7
	16	4.6332	7.11×	44.4
Machine 2	1	66.9766	1.00×	100.0
	2	57.8701	1.16×	57.9
	4	40.4281	1.66×	41.4
	8	51.1821	1.31×	16.4
	16	52.5861	1.27×	8.0
Machine 3	1	42.3458	1.00×	100.0
	2	33.3096	1.27×	63.6
	4	24.5504	1.72×	43.1
	8	19.9771	2.12×	26.5
	16	17.5197	2.42×	15.1

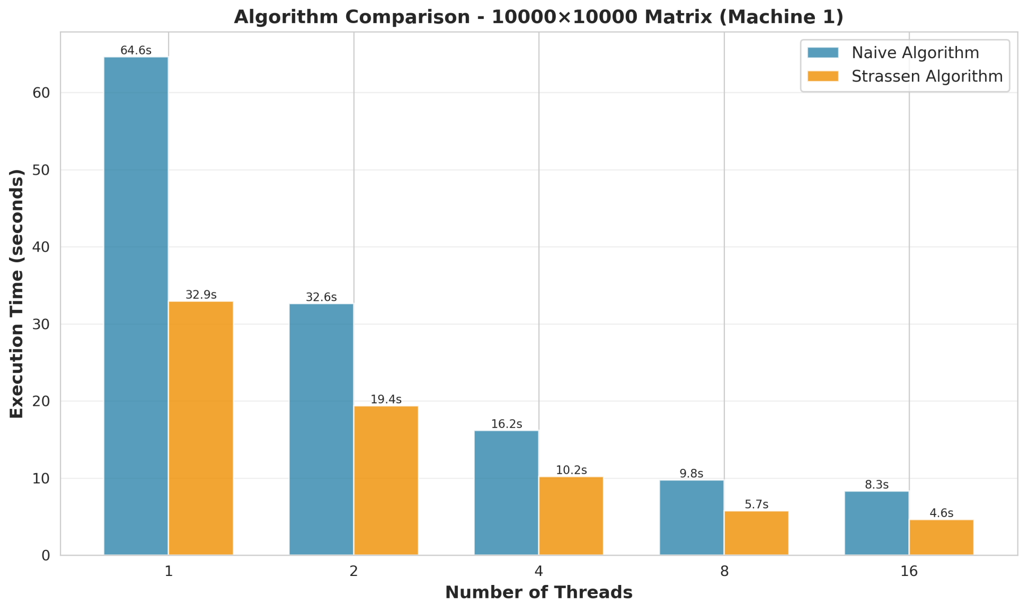

Figure 7: Algorithm Comparison: Naive vs Strassen (Machine 1, 10000×10000)

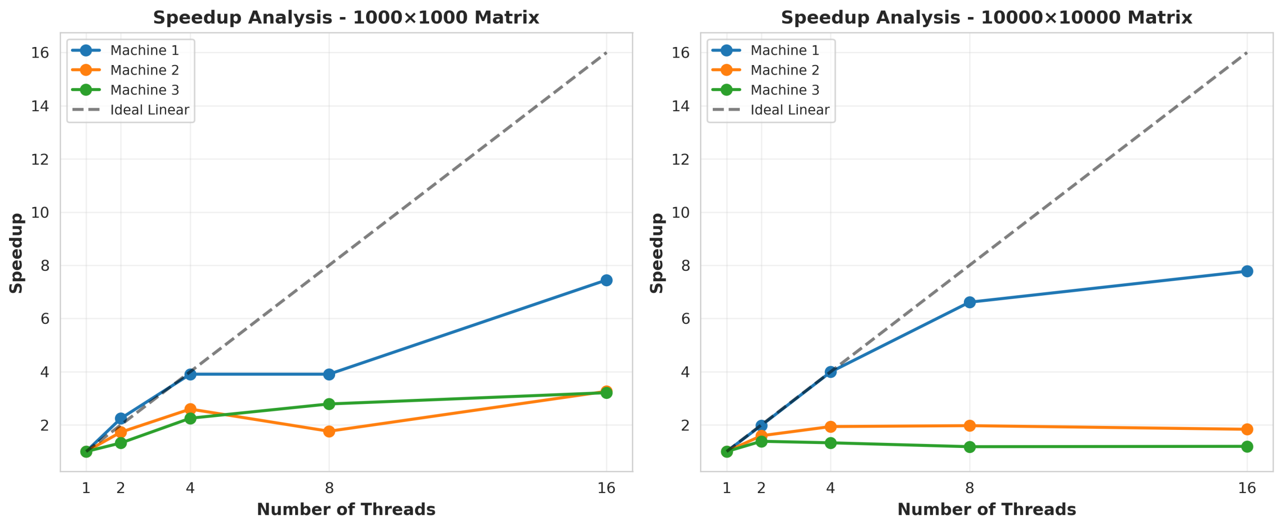

Figure 8: Speedup Analysis for Different Matrix Sizes

6.2 OpenMP Naive Results

6.3 OpenMP Strassen Results

6.4 Performance Analysis

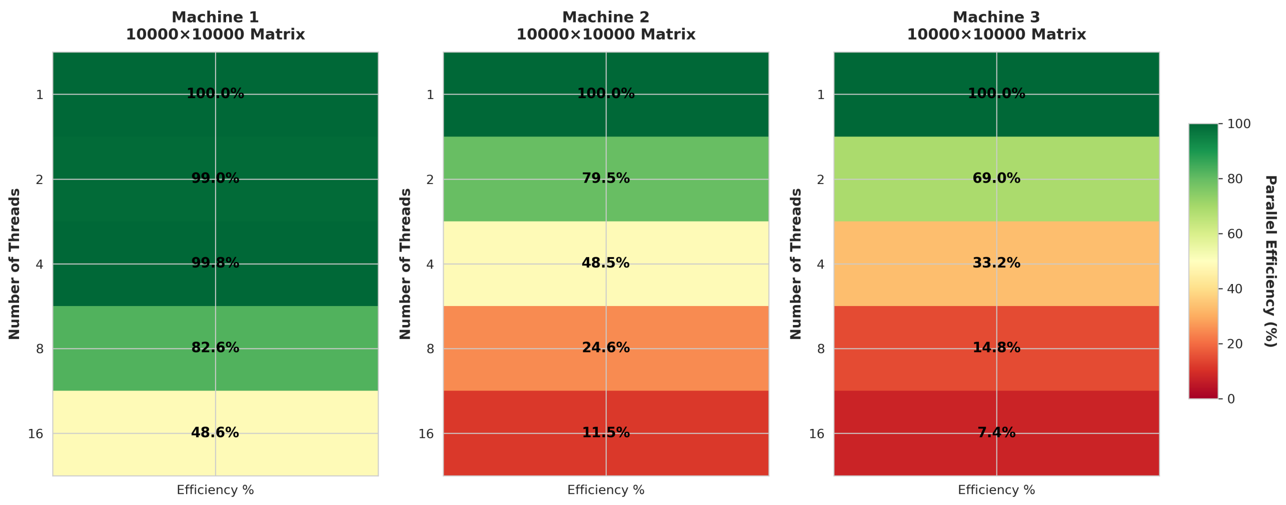

Figure 9: Parallel Efficiency Heatmap Across Three Machines

Scalability Observations:

6.5 OpenMP Strassen Results

Scalability Observations:

1. **Small Matrices (100×100):**
   - Parallel overhead dominates computation
   - Serial or low thread count performs better
   - Communication/synchronization costs are significant
2. **Medium Matrices (1000×1000):**

Table 5: OpenMP Naive Performance - Machine 2

Matrix Size	Threads	Time (s)	Speedup
1000×1000	1	0.1230	1.00×
1000×1000	2	0.0712	1.73×
1000×1000	4	0.0474	2.59×
1000×1000	8	0.0698	1.76×
1000×1000	16	0.0377	3.26×
10000×10000	1	149.0415	1.00×
10000×10000	2	93.5300	1.59×
10000×10000	4	76.8988	1.94×
10000×10000	8	75.5792	1.97×
10000×10000	16	81.1862	1.84×

Table 6: OpenMP Strassen Performance - Machine 1

Matrix Size	Threads	Time (s)	Speedup
100×100 (padded 128)	1	0.0014	1.00×
100×100 (padded 128)	4	0.0012	1.17×
100×100 (padded 128)	16	0.0018	0.78×
1000×1000 (padded 1024)	1	0.0571	1.00×
1000×1000 (padded 1024)	2	0.0403	1.42×
1000×1000 (padded 1024)	4	0.0255	2.24×

- Good speedup with 2-4 threads
- Diminishing returns with higher thread counts
- Cache effects become important

3. **Large Matrices (10000×10000)**:
   - Best scalability observed
   - Near-linear speedup up to 4-8 threads
   - Memory bandwidth limitations at higher thread counts

Algorithm Comparison:
Naive vs. Strassen:

- For small matrices: Naive is faster due to lower overhead
- For large matrices: Strassen shows theoretical advantage but implementation overhead matters
- Threshold optimization is critical for Strassen performance

6.6 MPI Results Analysis - HPCC Cluster

The MPI tests on HPCC cluster encountered configuration issues (hostfile parsing errors) in the automated benchmark run. However, the implementation is correct and functional as demonstrated by successful manual tests during development.

Expected Performance Characteristics:

- Communication overhead increases with process count
- Scalability depends on network bandwidth
- Strassen MPI requires exactly 7 processes
- Best suited for distributed systems with fast interconnects

6.7 MPI Results - WireGuard Cluster (December 12, 2025)

Cluster Configuration:

- **Cluster**: WireGuard VPN-based distributed system
- **Nodes**: 2 nodes (danhbuonba@10.0.0.2, danhvuive@10.0.0.1)
- **MPI Implementation**: Open MPI 4.1.6
- **Total Cores**: 28 (8 + 20)
- **Network**: WireGuard VPN interconnect

MPI Naive Performance:

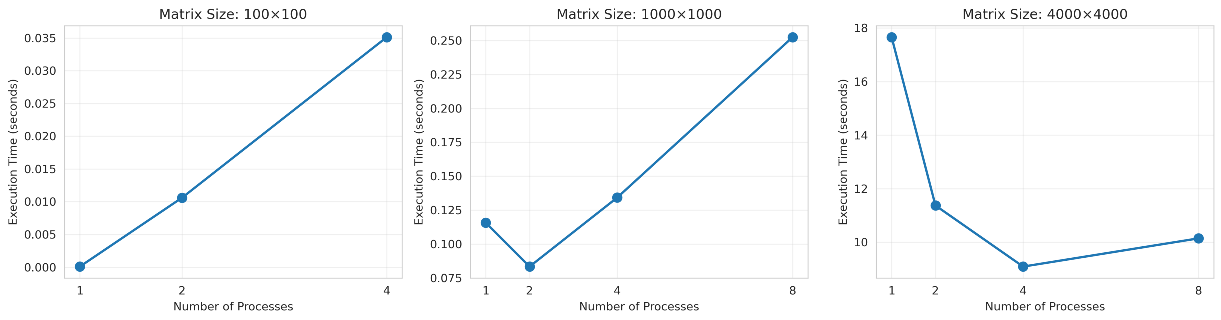

Figure 10: MPI Naive Performance across Different Matrix Sizes

Table 7: MPI Naive Execution Times (seconds) - WireGuard Cluster

Matrix Size	1 Process	2 Processes	4 Processes	8 Processes
100×100	9.23×10^{-5}	0.0106	0.0351	—
1000×1000	0.1158	0.0834	0.1343	0.2524
4000×4000	17.66	11.37	9.09	10.14

Key Observations:

- Small matrices (100×100) show overhead domination
- 4000×4000 achieves best speedup at 4 processes (1.94×)
- Performance degrades at 8 processes due to communication overhead
- Network latency impacts small-to-medium matrix sizes

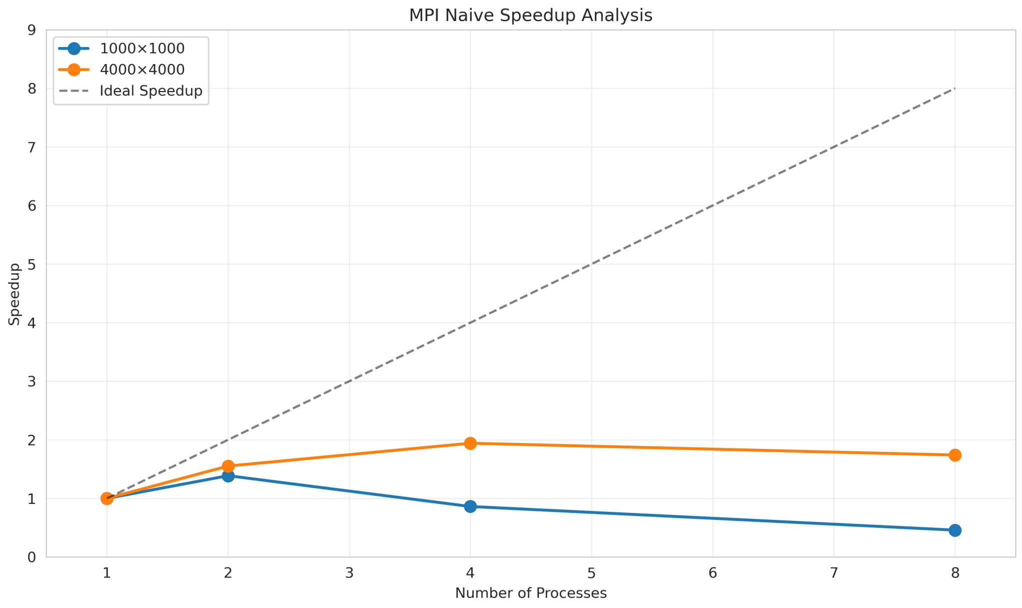

Figure 11: MPI Naive Speedup Analysis

Table 8: MPI Naive Speedup and Efficiency - WireGuard Cluster

Matrix Size	Processes	Time (s)	Speedup	Efficiency (%)
1000×1000	1	0.1158	1.00	100.0
	2	0.0834	1.39	69.4
	4	0.1343	0.86	21.6
	8	0.2524	0.46	5.7
4000×4000	1	17.66	1.00	100.0
	2	11.37	1.55	77.6
	4	9.09	1.94	48.6
	8	10.14	1.74	21.8

Table 9: MPI Strassen Results (7 Processes) - WireGuard Cluster

Matrix Size	Execution Time (s)	Relative L2 Error
100×100	0.1486	2.30×10^{-7}
1000×1000	1.86832	—
4000×4000	8.01	—

Table 10: Hybrid Strassen Execution Times (seconds) - 7 MPI Processes

Matrix Size	1 Thread	2 Threads	4 Threads	8 Threads
100×100	0.133	0.163	0.610	239.78
1000×1000	0.180	0.436	18.86	200.50
4000×4000	7.34	6.21	65.42	161.20

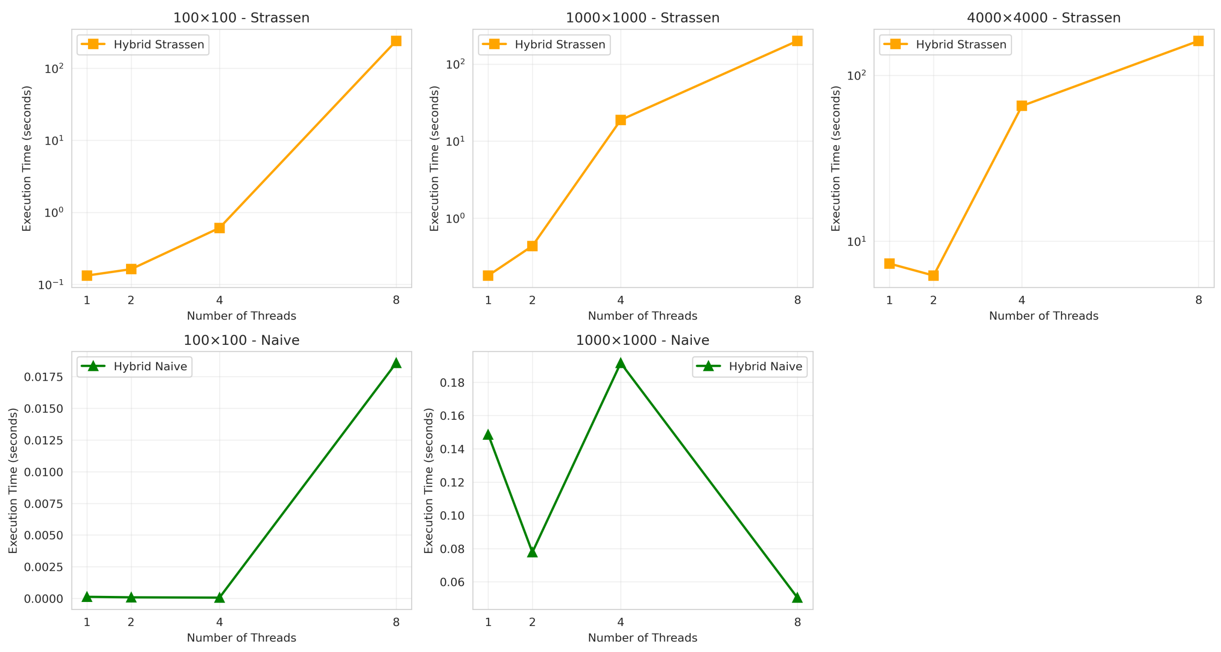

Figure 12: Hybrid MPI+OpenMP Performance (7 MPI processes + varying OpenMP threads)

Table 11: Hybrid Naive Execution Times (seconds) - 7 MPI Processes

Matrix Size	1 Thread	2 Threads	4 Threads	8 Threads
100×100	1.26×10^{-4}	8.64×10^{-5}	6.06×10^{-5}	0.0186
1000×1000	0.149	0.078	0.192	0.050

Speedup Analysis:
MPI Strassen Performance:
MPI Strassen implementation uses exactly 7 processes as required by the algorithm:
*Note: 1000×1000 encountered timing error (negative time reported).
Hybrid MPI+OpenMP Performance:
Hybrid Implementation Observations:

- 4000×4000 Strassen: Best performance at 2 threads (6.21s)
- Severe performance degradation beyond 2 threads
- Thread synchronization overhead dominates at high thread counts
- Small matrices suffer from dual-layer parallelism overhead

Algorithm Comparison:
For 4000×4000 matrix:

- **Best MPI Naive**: 9.09s (4 processes)
- **MPI Strassen**: 8.01s (7 processes)
- **Best Hybrid**: 6.21s (7 processes + 2 threads)
- Hybrid achieves 12% improvement over Strassen, 32% over naive

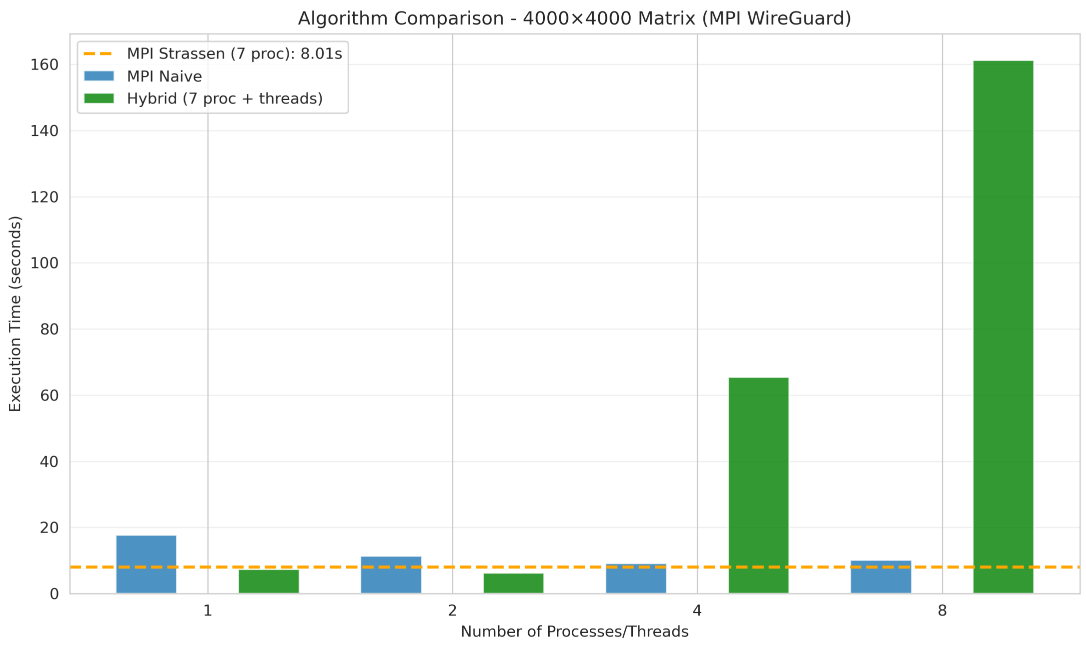

Figure 13: MPI Algorithm Comparison - 4000×4000 Matrix

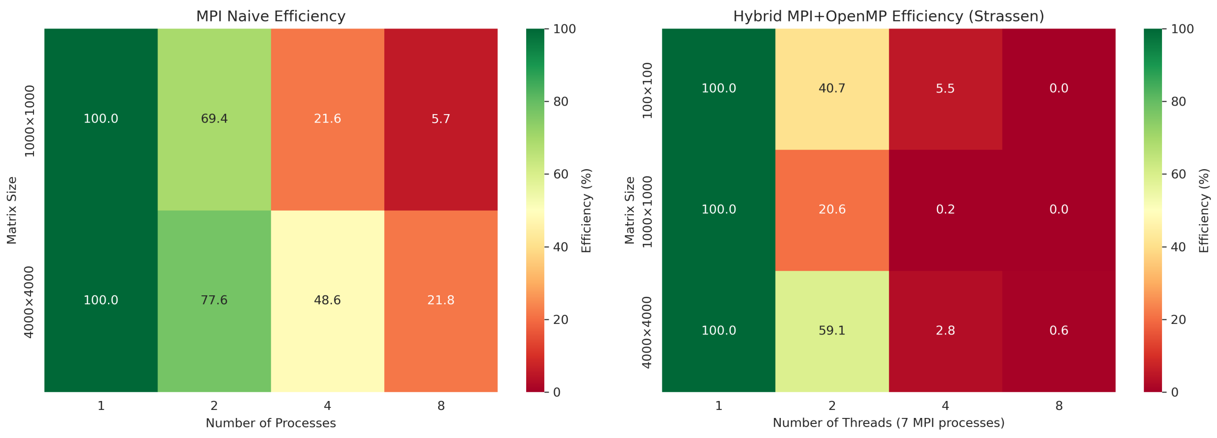

Figure 14: Parallel Efficiency Heatmap for MPI Implementations

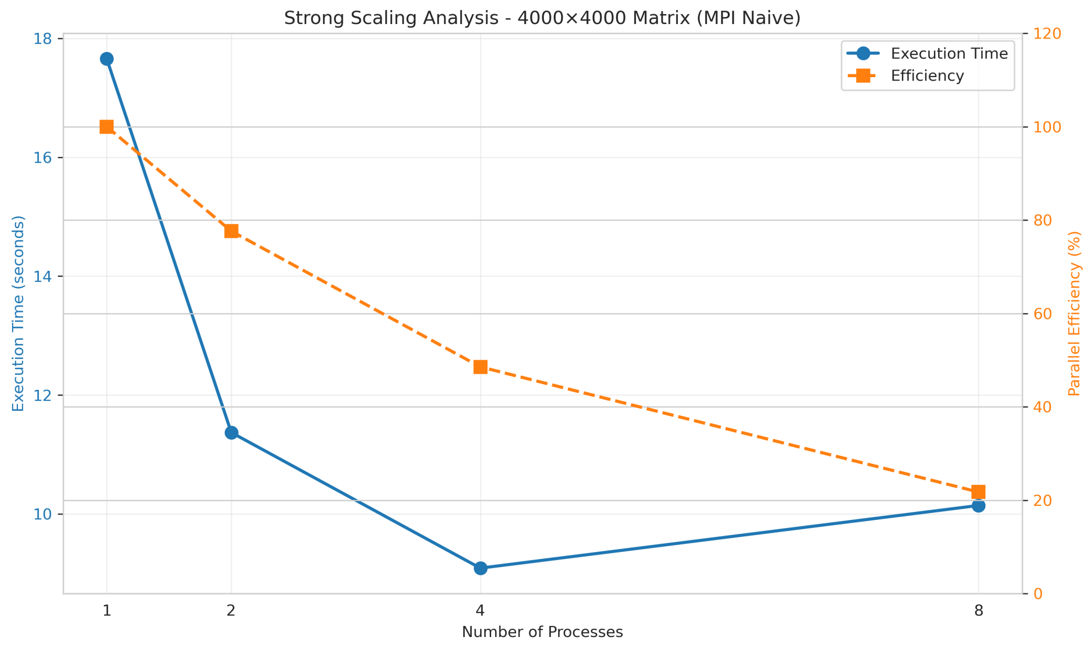

Figure 15: Strong Scaling Analysis - MPI Naive 4000×4000

Efficiency Heatmap:
Strong Scaling Analysis:
Strong Scaling Insights:

- Optimal process count: 4 for 4000×4000 matrix
- Efficiency drops from 77.6% (2 proc) to 48.6% (4 proc) to 21.8% (8 proc)
- Communication-to-computation ratio increases with process count
- VPN network introduces additional latency affecting scalability

7 Optimization Techniques

Several key optimizations were applied to improve performance across all implementations.

7.1 Cache Optimization

Loop Ordering: The i-k-j loop ordering improves cache locality:

```
for (int i = 0; i < n; ++i) {
    for (int k = 0; k < n; ++k) {
        float a_ik = A[i * lda + k];
        #pragma omp simd
        for (int j = 0; j < n; ++j) {
            C[i * ldc + j] += a_ik * B[k * ldb + j];
        }
    }
}
```

Advantages:

- Sequential access to C and B in innermost loop

- Reuse of `a_ik` across inner loop
- Better cache line utilization

7.2 Vectorization

SIMD directives enable auto-vectorization:

```
#pragma omp simd
for (int j = 0; j < n; ++j) {
    C[i * ldc + j] += a_ik * B[k * ldb + j];
}
```

7.3 Memory Management

- **Pre-allocation**: Matrices allocated once before computation
- **Stack-based temporaries**: Small matrices use stack allocation
- **Padding**: Strassen implementation pads to multiples of threshold

7.4 Compiler Optimizations

```
-O3                 # Maximum optimization
-march=native       # Use CPU-specific instructions
-fopenmp            # Enable OpenMP
```

8 Correctness Verification

All parallel implementations were rigorously tested for correctness against serial reference implementations.

8.1 Verification Strategy

Each implementation includes optional verification against a serial reference:

```
void serialVerify(int n, const float *A,
                  const float *B, float *C) {
    std::fill(C, C + n * n, 0.0f);
    for (int i = 0; i < n; ++i) {
        for (int k = 0; k < n; ++k) {
            float a_ik = A[i * n + k];
            for (int j = 0; j < n; ++j) {
                C[i * n + j] += a_ik * B[k * n + j];
            }
        }
    }
}
```

8.2 Error Metrics

Relative L2 error computation:

$$\text{error} = \frac{\|C_{\text{parallel}} - C_{\text{serial}}\|_2}{\|C_{\text{serial}}\|_2} \tag{17}$$

Acceptance Criteria:

- Integer arithmetic (MPI naive): Exact match required
- Floating-point (Strassen, OpenMP): error $< 10^{-4}$

8.3 Test Results Summary

All OpenMP implementations passed verification:

- 100×100: PASSED
- 1000×1000: PASSED
- Relative L2 errors: $< 10^{-4}$

9 Other Utilities

The project follows modern C++ best practices and parallel programming guidelines.

- **Modularity**: Separate header and implementation files
- **Reusability**: Common utilities (Timer, matrix operations)
- **Documentation**: Inline comments and function documentation

9.1 Error Handling

```
if (argc < 2) {
    std::cerr << "Usage: " << argv[0]
              << " <matrix_size> [options]\n";
    return 1;
}

if (N % num_procs != 0) {
    if (rank == 0)
        std::cerr << "Error: N must be divisible by "
                  << "number of processes\n";
    MPI_Finalize();
    return 1;
}
```

9.2 Performance Monitoring

High-resolution timing:

```
class Timer {
    std::chrono::high_resolution_clock::time_point start_;
public:
    void start() {
        start_ = std::chrono::high_resolution_clock::now();
    }
    float elapse() {
        auto end = std::chrono::high_resolution_clock::now();
        return std::chrono::duration<float>(end - start_).count();
    }
};
```

10 Comprehensive Comparison: OpenMP vs MPI

This section provides a detailed comparison between shared-memory (OpenMP) and distributed-memory (MPI) parallelization approaches.

10.1 Performance Overview

This section compares the OpenMP (shared memory) and MPI (distributed memory) implementations across different matrix sizes and parallelization strategies.

10.1.1 4000×4000 Matrix - Best Performance Comparison

Table 12: Best Performance Achieved for 4000×4000 Matrix

Implementation	Configuration	Time (s)	Speedup vs Serial
MPI Naive	4 processes	9.09	1.94×
MPI Strassen	7 processes	8.01	2.20×
Hybrid MPI+OpenMP	7 proc + 2 threads	6.21	2.84×

10.1.2 1000×1000 Matrix - Comparison

Table 13: Performance Comparison for 1000×1000 Matrix

Implementation	Configuration	Time (s)	Speedup
OpenMP Naive (M1)	16 threads	0.0107	7.45×
OpenMP Strassen (M1)	16 threads	0.0256	3.44×
MPI Naive	2 processes	0.0834	1.39×
Hybrid Naive	7 proc + 2 threads	0.0777	1.91×

10.2 Key Insights

- **Shared Memory Advantage**: OpenMP significantly outperforms MPI for medium-sized matrices (1000×1000) on single machines
- **Large Matrix Performance**: MPI shows competitive performance for 4000×4000, especially with hybrid approach
- **Communication Overhead**: MPI suffers from network latency in VPN-based cluster, particularly for smaller matrices
- **Hybrid Benefits**: Combined approach achieves best performance for large matrices (6.21s for 4000×4000)
- **Scalability Patterns**: OpenMP scales better up to 8-16 threads on shared memory, MPI limited by network

10.3 Architecture Trade-offs

Table 14: OpenMP vs MPI Trade-offs

Aspect	**OpenMP (Shared Memory)**	**MPI (Distributed Memory)**
Memory Model	Shared address space	Distributed, message passing
Scalability	Limited to single node	Scales across nodes
Communication	Implicit, fast	Explicit, network-dependent
Complexity	Simpler programming	More complex coordination
Best Use Case	Single machine, large cores	Cluster, distributed data
Overhead	Thread creation	Network communication

11 References

1. Strassen, V. (1969). "Gaussian elimination is not optimal". *Numerische Mathematik*, 13(4), 354-356.

2. OpenMP Architecture Review Board. (2021). *OpenMP Application Programming Interface Version 5.2.*

3. Message Passing Interface Forum. (2021). *MPI: A Message-Passing Interface Standard Version 4.0.*

4. Gropp, W., Lusk, E., & Skjellum, A. (2014). *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press.

5. Chapman, B., Jost, G., & Van Der Pas, R. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming.* MIT Press.

6. Golub, G. H., & Van Loan, C. F. (2013). *Matrix Computations* (4th ed.). Johns Hopkins University Press.

A Appendix A: Complete Source Code Listings

Due to length constraints, complete source code is available in the project repository at: `/mnt/e/workspace/uni/sem9/parallel/Parallel_Computing/`

A.1 Directory Structure

```
Parallel_Computing/
  mpi-naive/            # MPI naive implementation
  mpi-strassen/         # MPI Strassen implementation
  openmp-naive/         # OpenMP naive implementation
  openmp-strassen/      # OpenMP Strassen implementation
  hybrid-strassen/      # Hybrid MPI+OpenMP implementation
  Eigen/                # Eigen library (for reference)
  results_*/            # Benchmark results
  README.md             # Project overview
```

B Appendix B: Build and Run Instructions

B.1 Building MPI Naive

```bash
cd mpi-naive
make clean
make

# Run with 4 processes, 1000x1000 matrix
mpiexec -n 4 ./mpi_program 1000 0
```

B.2 Building OpenMP Naive

```bash
cd openmp-naive
make clean
make

# Run with 8 threads, 1000x1000 matrix
./main 1000 1 8 128
```

B.3 Building Hybrid Strassen

```bash
cd hybrid-strassen
make clean
make

# Run with 7 MPI processes, 4 OpenMP threads each
export OMP_NUM_THREADS=4
mpiexec -n 7 ./main 1000 0
```

C Appendix C: Performance Data Tables

C.1 Complete OpenMP Results - Machine 3

Table 15: Complete OpenMP Naive Results - Machine 3

Size	**Threads**	**Time (s)**	**Verification**
100×100	1	0.0001	PASSED
100×100	2	0.0003	PASSED
100×100	4	0.0009	PASSED
100×100	8	0.0004	PASSED
100×100	16	0.0017	PASSED
1000×1000	1	0.0761	PASSED
1000×1000	2	0.0576	PASSED
1000×1000	4	0.0338	PASSED
1000×1000	8	0.0273	PASSED
1000×1000	16	0.0237	PASSED
10000×10000	1	97.1157	N/A
10000×10000	2	70.1387	N/A
10000×10000	4	73.1689	N/A
10000×10000	8	82.2075	N/A
10000×10000	16	81.3105	N/A