Term Project - Multiplication Tables

Group 8

Name	Email	Student ID	Contribution
Mer Zhang	zhan5928@mylaurier.ca	169075928	Wrote the code and performed benchmark tests. Wrote the code for creating graphs.
Thooyavan Sundaralingam	sund6590@mylaurier.ca	210236590	Worked on the benchmark charts and graphs and helped with document/conclusion
Jacob Harper	harp0230@mylaurier.ca	201830230	Wrote problem statement and design section of report, helped with code. Wrote the code for creating graphs.
Cagri Isilak	isil4050@mylaurier.ca	210764050	Worked on the description of the graphs and wrote the conclusions
Shaun Bangalore	bang2000@mylaurier.ca	210912000	Worked on benchmark charts, statistic analysis and project report

Introduction

Problem

In this project, we develop a parallel algorithm using MPI (Message Passing Interface) to compute the number of distinct elements in an N×N multiplication table. The multiplication table is defined as an N×N array A, where each element is given by:

$$A_{ij} = i * j: 1 \leq i \leq j \leq N$$

Due to the table's symmetry, only elements above the main diagonal need to be considered for counting unique values. Elements in the upper triangle portion of the table may appear multiple times. In this project, we implement computation to determine the number of distinct values in a parallelized manner using MPI on the SCINET Teach cluster. This implementation minimizes computation time and distributes the workload evenly across multiple processes. The key challenge is handling the reduction of duplicate values while ensuring efficient communication and load balancing in the parallel environment. The deliverable of this project will be an MPI-based C program that computes the function: $M(N) =$ the number of distinct elements in an NxN matrix. Performance analysis and scalability testing will also be conducted to evaluate the efficiency of different parallel approaches we explored compared to a sequential implementation.

Design Considerations

Counting unique elements in multiplication tables requires data parallelism with a distributed memory architecture because we need to process different portions of an enormous computation space. We chose to use a distributed memory approach because the problem size quickly exceeds what any single machine can handle - for large N values, the memory required to track all potential products (up to N²) becomes prohibitive. Using a shared memory approach would create memory contention and limit scalability. Our distributed approach allows each process to handle a portion of the computation space independently, only communicating when necessary to merge results, which is crucial for handling problem sizes up to N = 10^5. Instead of sharing one massive array across all processes, each process maintains its segment of the solution space and communicates results only when necessary. The combination of data parallelism and distributed memory gives us the best performance for this computational problem, allowing us to calculate M(N) for extremely large N values.

Design Implementations

In this project, we explore a variety of implementations to solve the NxN multiplication table uniqueness problem. The following is a summary of each implementation as well as a discussion on the strengths and drawbacks of those implementations.

Basic Parallel Implementations

The algorithm computes the number of unique products from pairs of integers (i, j) where 1 ≤ i ≤ j ≤ N. In the sequential version, a boolean array is used to mark products that have already been seen, ensuring each unique product is only counted once. The basic parallel version distributes computations among MPI processes, having them check every pth row where p is the number of processes, each process computes its boolean array of seen values from 1 to n^2 then combines results using MPI_Reduce and MPI_Bcast. This design emphasizes cyclic work distribution, careful memory management, and synchronization to maintain consistency across processes.

Advanced Parallel Version with Segmented Bit Array, Cyclic Balancing

This version extends the previous implementation to handle large N by using a segmented bit array to manage memory consumption. It divides the total product space into smaller segments, and each MPI process computes products for its designated rows within a segment, applying an optimization that restricts j values to those producing products within the current segment. Our bit array approach uses memory-efficient techniques where each bit represents the presence or absence of a product. By using bit manipulation (with operations like byte |= (1 << bit)), we achieve an 8× memory reduction compared to using boolean arrays. Partial results are merged via a bitwise OR reduction, and unique product counts are tallied per segment before final aggregation and broadcast. This segmented approach enhances scalability and performance for very large input sizes.

Advanced Parallel Version with Segmented Bit Array, Block Distribution Balancing

This implementation is a permutation of the segmented bit array version discussed above but it uses a block distribution scheme, where each process is assigned a range of rows based on the formula $np = \lfloor n/P \rfloor + \{1\ if\ p < mod(n, P),\ 0\ else\}$. Like the segmented version, it restricts the j values to those yielding products within a given segment. Each process sets bits in its local array for computed products, and the arrays are merged using an MPI reduction with a bitwise OR operation, and the final unique count is broadcast. This implementation provides an alternative load balancing scheme to compare against the cyclic scheme.

Advanced Parallel Version with Segmented Bit Array, Using Cannon's Algorithm

This version employs Cannon's algorithm to partition the computation over a two-dimensional, square MPI process grid. Each process is assigned a block of the input space, and segmented

bit arrays are used to manage the large product space efficiently. Through cyclic shifts implemented via dedicated row and column communicators, processes iteratively exchange block boundaries to cover all necessary combinations while preserving the upper-triangular structure. The segmented results are then merged with a bitwise OR reduction, and the unique product count is computed and broadcast. This method is well-suited for environments with a perfect-square number of processes and emphasizes both scalability and efficient memory usage.

First Factor Based Memory Optimized Implementation
This approach introduces a first factor function that checks whether (i, j) is the minimal factor representation for the product, ensuring that each product is counted only once. The optimized parallel version distributes rows among MPI processes cyclically and applies the first factor check to filter out duplicate counts. The results are aggregated using MPI_Reduce and MPI_Bcast, parallelizing the sequential verification method while maintaining correctness through the first factor checking.

Cannon's Algorithm with Processed Flag Array for Double Counting Prevention
This function implements Cannon's algorithm to compute M(N) using a 2D MPI grid where each process handles a specific block of the (i, j) product space. In addition to using a first factor check, each process maintains a flag array to track processed pairs, preventing double counting during cyclic shifts. Processes exchange block boundaries with their neighbors using MPI_Sendrecv, and the overall count is aggregated via MPI_Allreduce. Finally, the result is adjusted (halved) to account for the symmetry in factor pairs, ensuring an accurate count.

Design Comparisons

In comparing these implementations, memory management and workload distribution are the primary factors we are concerned about. The basic parallel implementations use boolean arrays, which are effective for smaller N but become less efficient as N grows. During testing we found that it cannot process values above N = 4000. In contrast, the advanced versions leverage segmented bit arrays to reduce memory overhead and scale efficiently for large N values. Lastly, the first-factor-based method effectively uses no memory, which should scale much better than the segmented bit array method in terms of memory management. For our load-balancing algorithms, the basic parallel version distributes rows cyclically. Cyclic distribution assigns rows to processes in a round-robin fashion, providing better load balancing when the work per row varies significantly. Block distribution assigns contiguous ranges of rows to each process, which improves data locality but can lead to load imbalance as processes handling higher-numbered rows perform fewer computations. Lastly, the Cannon's algorithm-based implementations adopt a two-dimensional process grid to balance workload more evenly, however, the process count is a **perfect square**. While Cannon’s algorithm theoretically offers a balanced 2D decomposition, it has a higher communication overhead due to the cyclic shifts required between computation phases. The use of first factor checks across

the last two implementations ensures that duplicate counting is minimized, with one version further safeguarding against double counting through the use of a processed flag array during cyclic shifts. Theoretically , the segmented approaches and those based on Cannon's algorithm should demonstrate superior scalability and performance for large values of N, though they require more complex communication patterns and synchronization. The choice among these methods ultimately depends on the problem size, available computing resources, and specific performance requirements of the project. We decided that benchmarking the results from each method against the increasing values of N and for each number of processors, would be the best way to decide which algorithm to settle on.

Verifying by hand that M(5) = 14 and M(10) = 42:

5 x 5 Mult Table

	1	2	3	4	5
1	1	2	3	4	5
2		4	6	8	10
3			9	12	15
4				16	20
5					25
			Total: Distinct = 14		

10 x 10 Mult Table

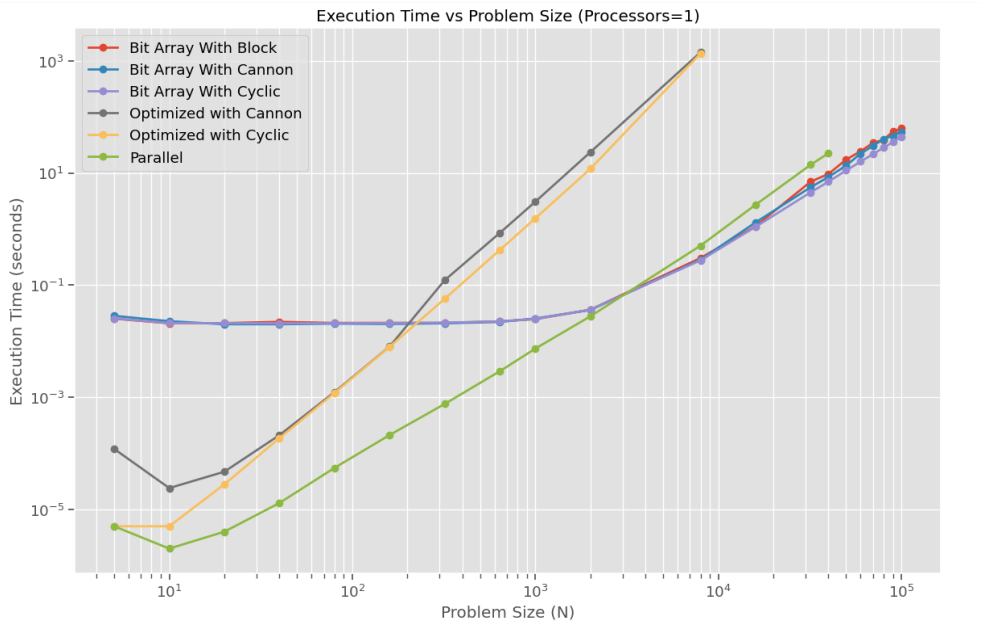
	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2		4	6	8	10	12	14	16	18	20
3			9	12	15	18	21	24	27	30
4				16	20	24	28	32	36	40
5					25	30	35	40	45	50
6						36	42	48	54	60
7							49	56	63	70
8								64	72	80
9									81	90
10										100
								Total: Distinct = 42		

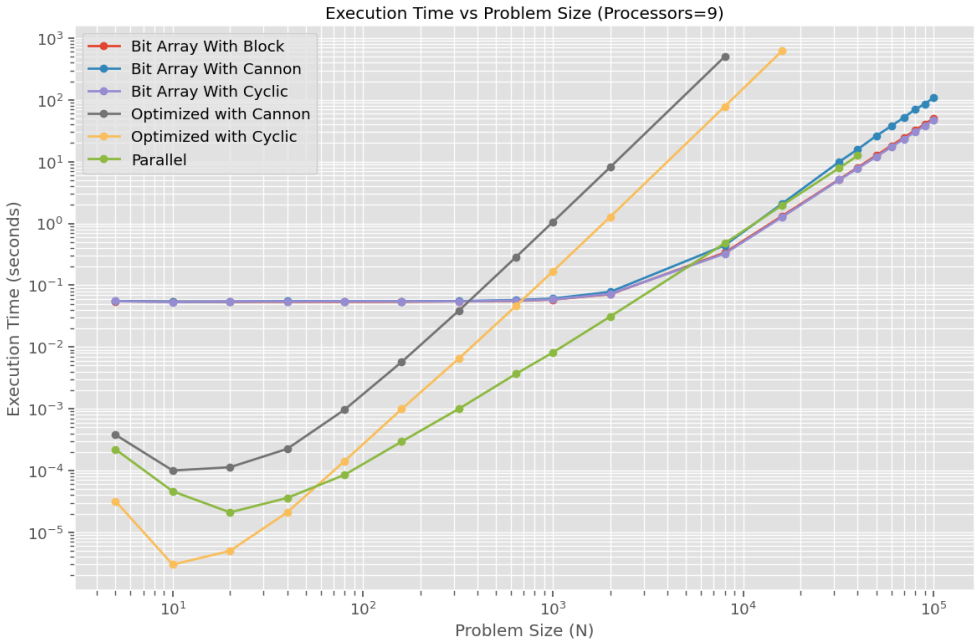
Results:

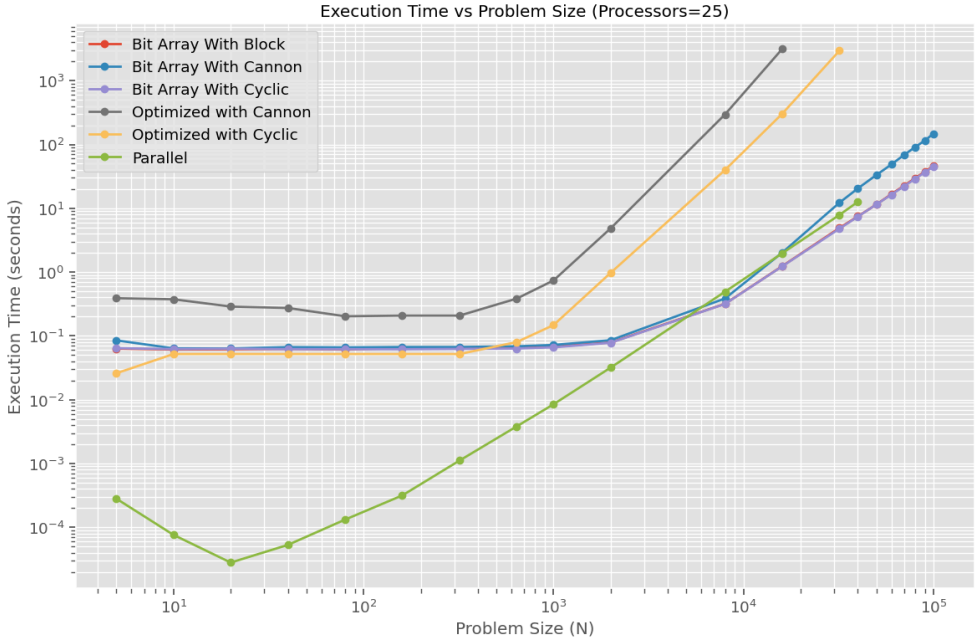
20	40	80	160	320	640	1,000	2,000	8,000	16,000	32,000
152	517	1,939	7,174	27,354	27,354	248,083	959,759	1,450,9549	56,705,617	221,824,366

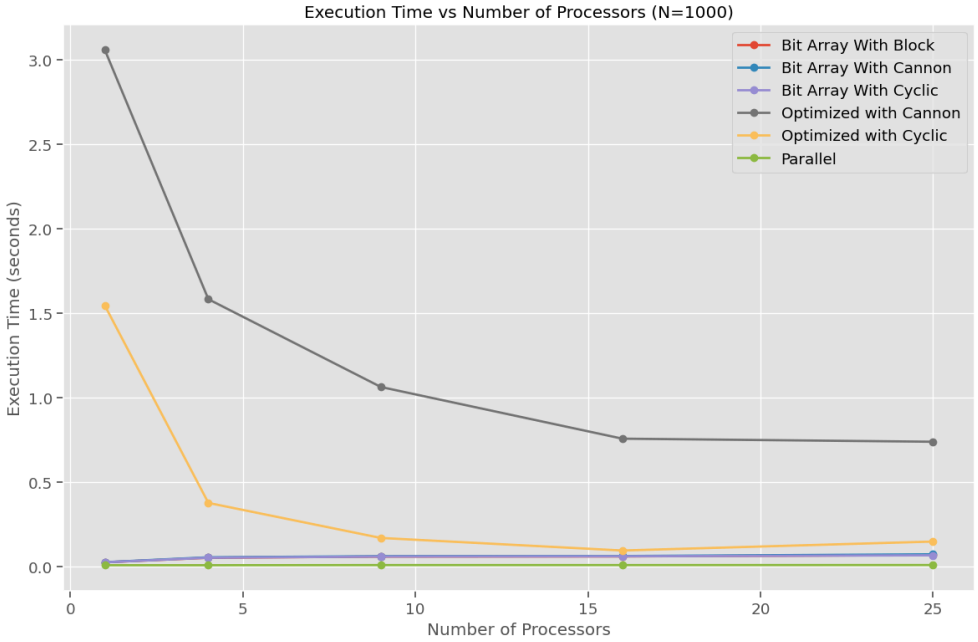
40,000	50,000	60,000	70,000	80,000	90,000	100,000
344,462,009	534,772,334	766,265,795	1,038,159,781	1,351,433,197	1,704,858,198	2,099,198,630

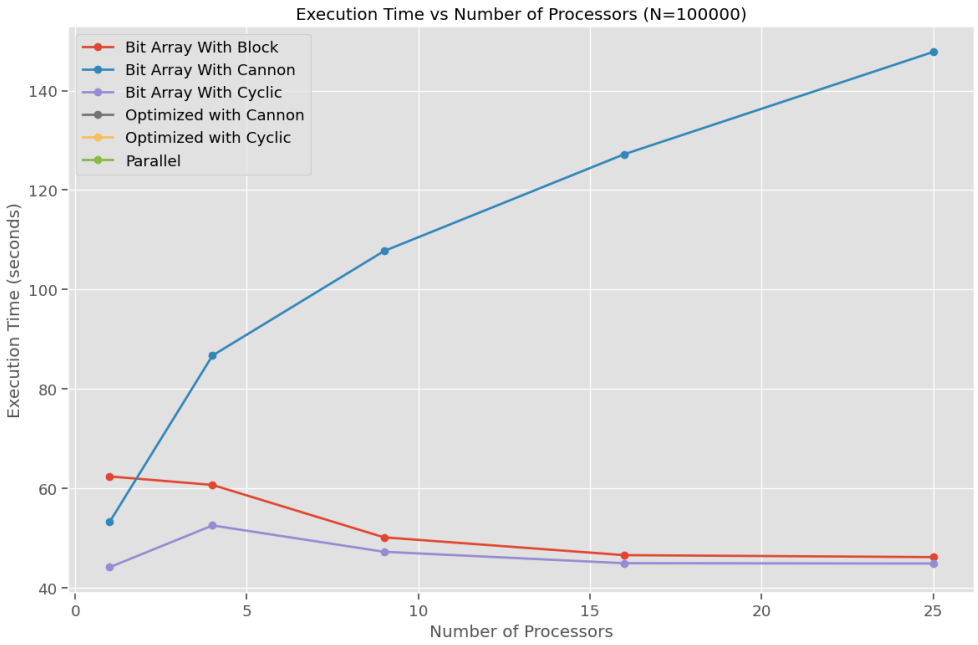
Performance Trends:

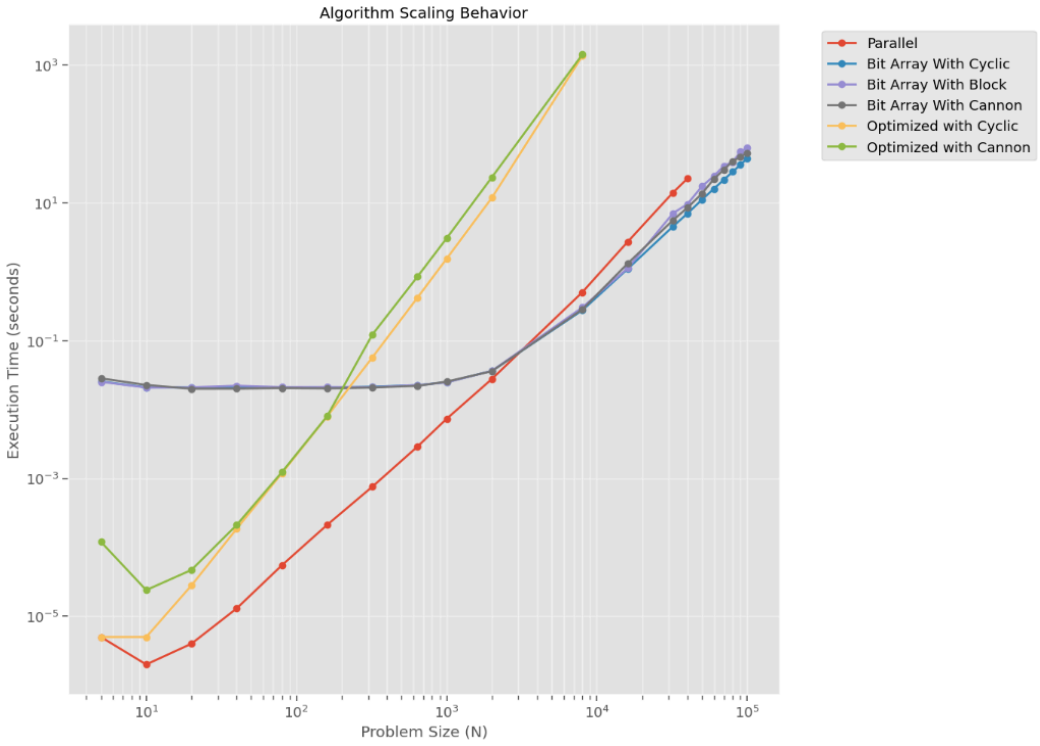





Execution Time vs Problem Size (Processors=25)
Bit Array With Block
Bit Array With Cannon
Bit Array With Cyclic
Optimized with Cannon
Optimized with Cyclic
Parallel
Execution Time (seconds)
Problem Size (N)

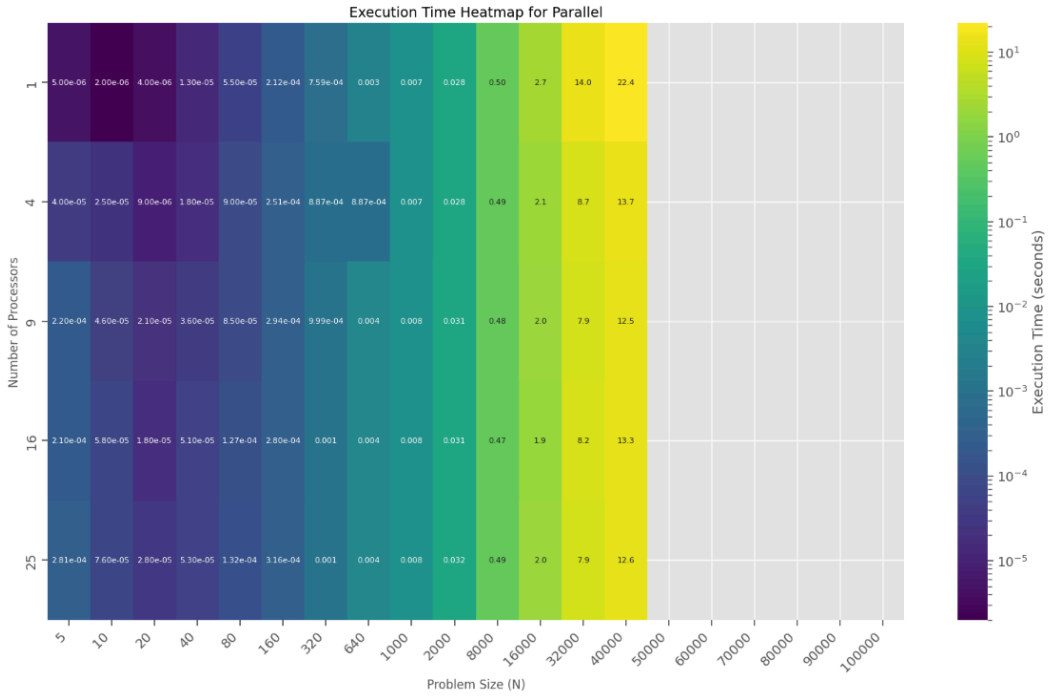

Execution Time vs Number of Processors (N=1000)
Bit Array With Block
Bit Array With Cannon
Bit Array With Cyclic
Optimized with Cannon
Optimized with Cyclic
Parallel
Execution Time (seconds)
Number of Processors



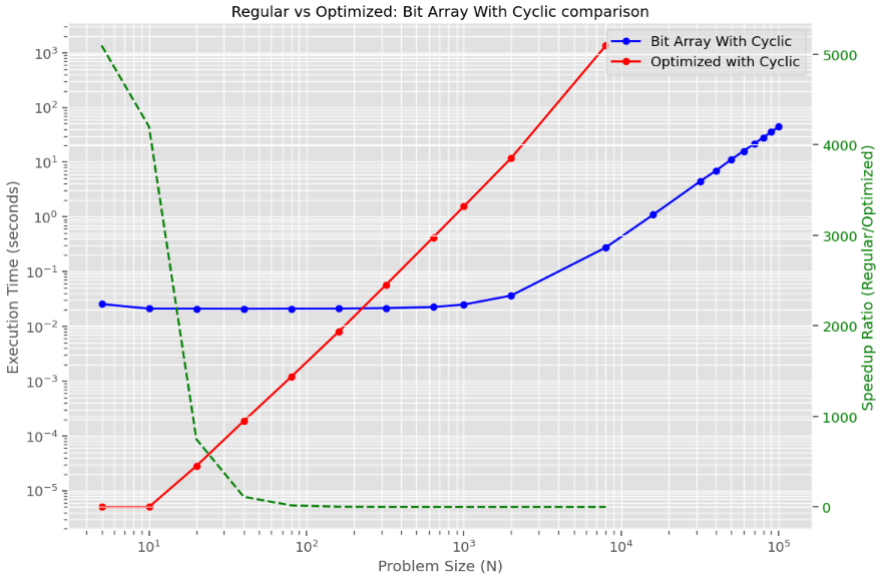


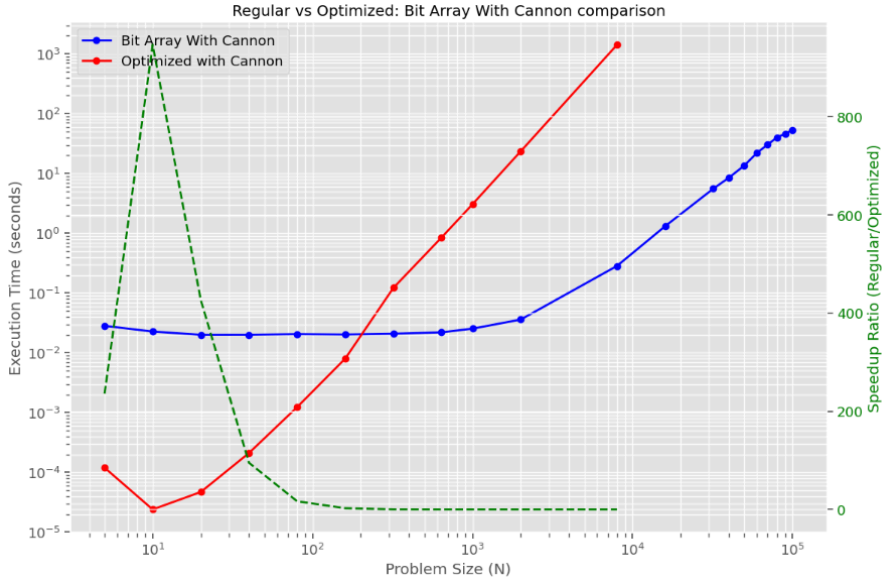
The above graphs show how each algorithm's execution time grows as the size of the problem increases. The standard parallel implementation hits memory constraints around N=4000, unable to complete for larger values, so the graph shows how it is expected to grow without the memory constraints. While memory optimized algorithms scale beyond this limit, they don't match the efficiency of bit array with Block/Cyclic methods for large N values. Memory optimized

with Cannon frequently fails to complete large N, primarily due to its higher communication cost as processes must exchange boundaries during each shift phase. The bit array implementations provide the best overall balance between memory efficiency and computational performance across the full range of N values tested.

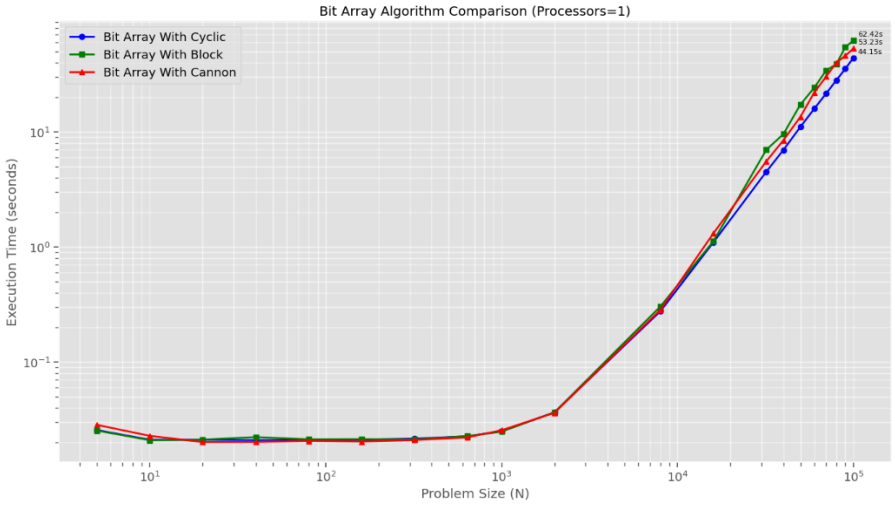


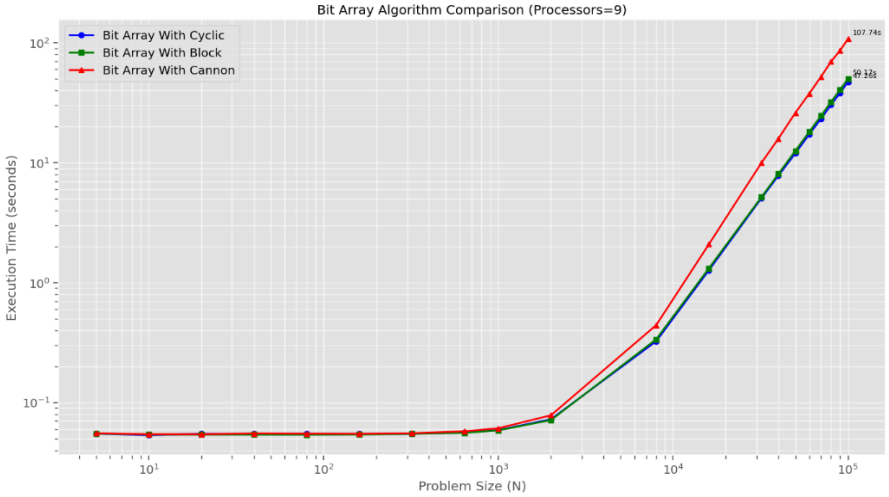
The heat map of execution time shows execution time across processors and N values. The low times shown by dark colours are concentrated in lower N values and lower processor counts, while the lighter colours show higher execution time or DNF. Some algorithms show flat heat patterns, meaning they are less sensitive to processor count, which usually means it bottlenecks or has poor scalability. Optimized methods often have blank or light coloured areas, and cannon's algorithm has light and uneven patterns, showing it's the least efficient under high concurrency.

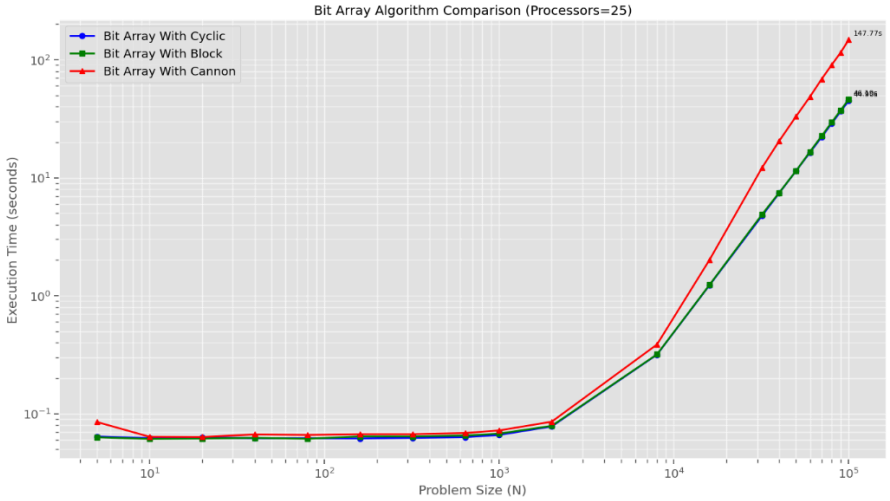


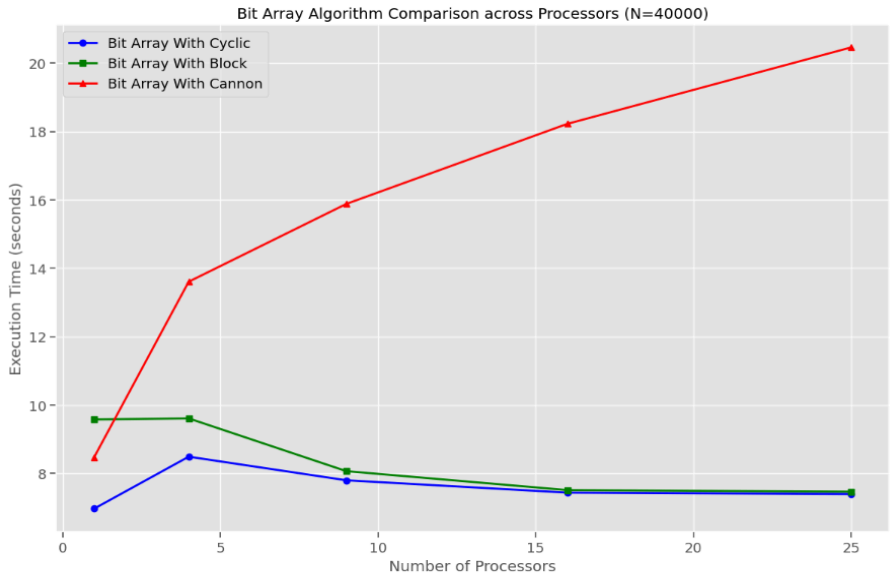


These graphs show the execution of regular vs optimized implementations and the speedup ratio plotted on a second y-axis. Optimized versions outperformed their counterparts by orders of magnitude for smaller N. At higher N, optimized versions tend to fail due to memory, thread contention or algorithm limits.


Bit Array Algorithm Comparison (Processors=1)
Bit Array With Cyclic
Bit Array With Block
Bit Array With Cannon
62.42s
53.23s
44.15s
Execution Time (seconds)
Problem Size (N)


Bit Array Algorithm Comparison (Processors=9)
Bit Array With Cyclic
Bit Array With Block
Bit Array With Cannon
107.74s
Execution Time (seconds)
Problem Size (N)


Bit Array Algorithm Comparison (Processors=25)
Bit Array With Cyclic
Bit Array With Block
Bit Array With Cannon
147.77s
Execution Time (seconds)
Problem Size (N)



The above graphs further examine the differences in performance caused by different load-balancing methods. Using the segmented bit array solution we found that cyclic load balancing performs marginally better than block balancing across all N values and Cannon underperforms both options. Moreover, both block and cyclic find noticeable improvements as the number of processors increase where Cannon seems to lose efficiency with greater numbers of processes likely due to heavy communication overhead.

Limit Testing

Using the fastest algorithm (segmented bit array with cyclic load balancing), we tested larger N values using 9 processors. Here are the final results:

Processors	M(200,000) = 8,250,516,470	M(500,000) = 50,474,982,416	M(750,000) = 112,556,022,471
9	192.341122s	1151.285162s	2689.310988s

Conclusion

In this project, we tried out many design implementations using MPI based parallel algorithms to see how they would fare in computing the number of distinct elements in a NxN multiplication table. These included basic parallel implementations, segmented bit array implementations with cyclic balancing, block distribution balancing and Cannon's algorithm, a first factor based memory optimized implementation, and an optimized Cannon's algorithm with a processed flag array. The key evaluation metrics were execution time, scalability, speedup and how each algorithm handles communication overhead as N and processor count increases. In testing, we found that standard parallel hits a memory constraint around N = 4000, but before that point has the fastest execution time. Memory optimized algorithms were able to compute past this, but they were beaten in efficiency by the bit array implementations with block distribution and cyclic balancing. In comparing the bit array and optimized implementations, we found that in the smaller N values, optimized outperformed, but as N increased, they tended to fail due to memory or thread contention limits. Although we hypothesized that bit array with Cannon's algorithm should in theory work the best, it did not in practice likely due to a higher communication overhead. Thus our findings show that for small N the best is the standard parallel implementation, and for a large N, the best implementation for scalability is the segmented bit array with cyclic balancing. Overall, the best algorithm is the segmented bit array with cyclic balancing in terms of the key evaluation metrics stated. Although it loses out to the standard parallel implementation for small N, the difference in execution time at that N range is negligible, and cyclic has the best scalability for large N values, as it scales the best with processor count.

Code:

standard _parallel_cyclic.c

```
/**
 * Title: Parallel Processing Assignment - Compute Unique Products M(N)
 * ------------------------------------------------------------------------------------
 * Description: This program computes the number of unique products (i * j) for 1 <= i <= j <= N.
 *              It uses MPI for parallelization, simple array-based
 * Author:
 * - Mer Zhang 169075928
 * - Thooyavan Sundaralingam 210236590
 * - Jacob Harper 201830230
 * - Cagri Isilak 210764050
 * - Shaun Bangalore 210912000
 */
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <math.h>
#include <mpi.h>

// Sequential implementation for verification
int compute_M_N_sequential(int N) {
    bool *exists = (bool *)calloc((N * N) + 1, sizeof(bool));
    if (!exists) {
        printf("Failed in sequential\n");
        return -1;
    }

    // Nested loop from 1 <= i <= j <= N check i*j
    int count = 0;
    for (int i = 1; i <= N; i++) {
        for (int j = i; j <= N; j++) {
            int prod = i * j;
            if (!exists[prod]) {
                exists[prod] = true;
                count++;
            }
        }
    }
    free(exists);
    return count;
}

// Parallel version
int compute_M_N_parallel(int N) {
    // Set up MPI variables
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // Create N^2 array to check values against
    long long max_prod = (long long)N * N;
    char *local_seen = (char *)calloc(max_prod + 1, sizeof(char));
    if (!local_seen) {
```

```
        MPI_Abort(MPI_COMM_WORLD, 1);
        return -1;
    }

    // Each process reports its row range
    int first_row = 1 + rank;
    int last_row = N;
    int approx_rows = (N - first_row) / size + 1;
    int rows_processed = 0;

    // Nested loop for i = my_rank * x <= N and i <= j <= N check i*j
    for (int i = first_row; i <= N; i += size) {
        for (int j = i; j <= N; j++) {
            long long prod = (long long)i * j;
            local_seen[prod] = 1;
        }
        rows_processed++;
    }

    // Synchronize to ensure all processes have completed their computation
    MPI_Barrier(MPI_COMM_WORLD);
    // Merge all process arrays to central array
    char *global_seen = NULL;
    if (rank == 0) {
        global_seen = (char *)calloc(max_prod + 1, sizeof(char));
        if (!global_seen) {
            MPI_Abort(MPI_COMM_WORLD, 2);
            return -1;
        }
    }

    MPI_Reduce(local_seen, global_seen, max_prod + 1, MPI_CHAR, MPI_MAX, 0, MPI_COMM_WORLD);
    // Count all distinct values
    int result = 0;
    if (rank == 0) {
        for (long long i = 1; i <= max_prod; i++) {
            if (global_seen[i]) result++;
        }
        free(global_seen);
    }
```

```
    free(local_seen);

    // Broadcast the result to all processes
    MPI_Bcast(&result, 1, MPI_INT, 0, MPI_COMM_WORLD);

    return result;
}

int main(int argc, char **argv) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (argc < 2) {
        if (rank == 0) {
            printf("Usage: %s <N1> [N2 N3 ...]\n", argv[0]);
        }
        MPI_Finalize();
        return 1;
    }

    if (rank == 0) {
        printf("Verifying known results (sequential):\n");
        int m5 = compute_M_N_sequential(5);
        printf("M(5) = %d (Expected: 14)\n", m5);
        int m10 = compute_M_N_sequential(10);
        printf("M(10) = %d (Expected: 42)\n\n", m10);
    }

    for (int i = 1; i < argc; i++) {
        int N = atoi(argv[i]);
        if (N <= 0) {
            if (rank == 0) {
                printf("WARNING: Skipping invalid N value: %s\n", argv[i]);
            }
            continue;
        }
```

```
        if (rank == 0) {
            printf("\n========================================\n");
            printf("Computing M(%d) using %d processes...\n", N, size);
            printf("========================================\n");
        }

        double t_start = MPI_Wtime();

        int result;
        result = compute_M_N_parallel(N);

        double t_end = MPI_Wtime();

        if (rank == 0) {
            printf("\nRESULT: M(%d) = %d\n", N, result);
            printf("Time taken: %.6f seconds\n", t_end - t_start);
            printf("========================================\n\n");
        }
    }

    MPI_Finalize();
    return 0;
}
```

cyclic-bit-array.c

```
/**
 * Title: Parallel Processing Assignment - Compute Unique Products M(N)
 * ---------------------------------------------------------------------------------------
 * Description: This program computes the number of unique products (i * j) for 1 <= i <= j <= N.
 *              Using advanced bit array segmentation with cyclic balancing.
 */

// Sequential implementation for verification
int compute_M_N_sequential(int N) {
    bool *exists = (bool *)calloc((N * N) + 1, sizeof(bool));
    if (!exists) {
        printf("Failed in sequential\n");
```

```
        return -1;
    }

    // Nested loop from 1 <= i <= j <= N check i*j
    int count = 0;
    for (int i = 1; i <= N; i++) {
        for (int j = i; j <= N; j++) {
            int prod = i * j;
            if (!exists[prod]) {
                exists[prod] = true;
                count++;
            }
        }
    }
    free(exists);
    return count;
}
// Advanced parallel version for large N using segmented bit array
long long compute_M_N_advanced(int N) {
    // Set up MPI variables
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Setup for segmented bit array processing
    long long max_prod = (long long)N * N;
    const long long SEGMENT_SIZE = 100000000LL; // 100M elements per segment
    long long num_segments = (max_prod / SEGMENT_SIZE) + 1;

    unsigned char *segment = (unsigned char *)calloc((SEGMENT_SIZE / 8) + 1, sizeof(unsigned
char));

    // Error handling: failed to create segment
    if (!segment) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return -1;
    }

    long long total_unique = 0;
```

```
    for (long long s = 0; s < num_segments; s++) {
        // Clear segment buffer
        memset(segment, 0, (SEGMENT_SIZE / 8) + 1);

        // Define range for this segment
        long long min_p = s * SEGMENT_SIZE + 1;
        long long max_p = (s + 1) * SEGMENT_SIZE;
        if (max_p > max_prod) max_p = max_prod;

        // Number of rows each process will handle
        int rows_per_proc = (N / size) + (rank < (N % size) ? 1 : 0);

        int rows_processed = 0;
        for (int i = 1 + rank; i <= N; i += size) {
            // Optimization: only consider j values that produce products in this segment
            long long min_j = min_p / i;
            if (min_j < i) min_j = i;  // respect symmetry

            long long max_j = max_p / i;
            if (max_j > N) max_j = N;

            for (long long j = min_j; j <= max_j; j++) {
                long long prod = i * j;
                if (prod >= min_p && prod <= max_p) {
                    long long offset = prod - min_p;
                    long long byte = offset / 8;
                    int bit = offset % 8;
                    segment[byte] |= (1 << bit);
                }
            }
            rows_processed++;
        }

        // Synchronize before reduction
        MPI_Barrier(MPI_COMM_WORLD);

        unsigned char *merged = NULL;
        if (rank == 0) {
            merged = (unsigned char *)calloc((SEGMENT_SIZE / 8) + 1, sizeof(unsigned char));
            if (!merged) {
```

```
                MPI_Abort(MPI_COMM_WORLD, 3);
                return -1;
            }
        }

        MPI_Reduce(segment, merged, (SEGMENT_SIZE / 8) + 1, MPI_UNSIGNED_CHAR, MPI_BOR, 0,
MPI_COMM_WORLD);

        if (rank == 0) {
            long long segment_count = 0;
            for (long long i = 0; i < (SEGMENT_SIZE / 8) + 1; i++) {
                unsigned char byte = merged[i];
                while (byte) {
                    if (byte & 1) segment_count++;
                    byte >>= 1;
                }
            }

            total_unique += segment_count;
            free(merged);
        }
    }

    free(segment);

    // Broadcast the final result to all processes
    MPI_Bcast(&total_unique, 1, MPI_LONG_LONG_INT, 0, MPI_COMM_WORLD);

    return total_unique;
}

int main(int argc, char **argv) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (argc < 2) {
        if (rank == 0) {
            printf("Usage: %s <N1> [N2 N3 ...]\n", argv[0]);
```

```
        }
        MPI_Finalize();
        return 1;
    }

    if (rank == 0) {
        printf("Verifying known results (sequential):\n");
        int m5 = compute_M_N_sequential(5);
        printf("M(5) = %d (Expected: 14)\n", m5);
        int m10 = compute_M_N_sequential(10);
        printf("M(10) = %d (Expected: 42)\n\n", m10);
    }

    for (int i = 1; i < argc; i++) {
        int N = atoi(argv[i]);
        if (N <= 0) {
            if (rank == 0) {
                printf("WARNING: Skipping invalid N value: %s\n", argv[i]);
            }
            continue;
        }

        if (rank == 0) {
            printf("\n========================================\n");
            printf("Computing M(%d) using %d processes...\n", N, size);
            printf("========================================\n");
        }

        double t_start = MPI_Wtime();

        int result;

        result = compute_M_N_advanced(N);

        double t_end = MPI_Wtime();

        if (rank == 0) {
            printf("\nRESULT: M(%d) = %lld\n", N, result);
            printf("Time taken: %.6f seconds\n", t_end - t_start);
            printf("========================================\n\n");
```

```
        }
    }

    MPI_Finalize();
    return 0;
}
```

block-bit-array.c

```c
/**
 * Title: Parallel Processing Assignment - Compute Unique Products M(N)
 * ---------------------------------------------------------------------------------------
 * Description: This program computes the number of unique products (i * j) for 1 <= i <= j <= N.
 *              Using advanced bit array segmentation with block distribution.
 */

// Sequential implementation for verification
int compute_M_N_sequential(int N) {
    bool *exists = (bool *)calloc((N * N) + 1, sizeof(bool));
    if (!exists) {
        printf("Failed in sequential\n");
        return -1;
    }

    // Nested loop from 1 <= i <= j <= N check i*j
    int count = 0;
    for (int i = 1; i <= N; i++) {
        for (int j = i; j <= N; j++) {
            int prod = i * j;
            if (!exists[prod]) {
                exists[prod] = true;
                count++;
            }
        }
    }
    free(exists);
    return count;
}

// Advanced parallel version with block distribution and bit array segmentation
```

```
long long compute_M_N_block_advanced(int N) {
    // Set up MPI variables
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Calculate block distribution using the formula
    // np = ⌊n/P⌋ + {1 if p < mod(n, P), 0 else}
    int base_rows = N / size;
    int extra = N % size;
    int rows_for_rank = base_rows + (rank < extra ? 1 : 0);
    // +1 because rows start at 1
    int start_row = rank * base_rows + (rank < extra ? rank : extra) + 1;
    int end_row = start_row + rows_for_rank - 1;

    if (rank == 0) {
        printf("Process %d handles rows %d to %d (%d rows)\n", rank, start_row, end_row,
rows_for_rank);
    }
    // Instantiate segments
    long long max_prod = (long long)N * N;
    const long long SEGMENT_SIZE = 100000000LL; // 100M elements per segment
    long long num_segments = (max_prod / SEGMENT_SIZE) + 1;
    unsigned char *segment = (unsigned char *)calloc((SEGMENT_SIZE / 8) + 1, sizeof(unsigned
char));
    // Error handling, failed to generate segment
    if (!segment) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return -1;
    }

    long long total_unique = 0;

    // Process each segment of the product space
    for (long long s = 0; s < num_segments; s++) {
        // Clear segment buffer
        memset(segment, 0, (SEGMENT_SIZE / 8) + 1);

        // Define range for this segment
        long long min_p = s * SEGMENT_SIZE + 1;
```

```
        long long max_p = (s + 1) * SEGMENT_SIZE;
        if (max_p > max_prod) max_p = max_prod;

        // Block distribution - each process handles a contiguous range of rows
        for (int i = start_row; i <= end_row; i++) {
            // Optimization: only consider j values that produce products in this segment
            long long min_j = min_p / i;
            if (min_j < i) min_j = i;  // respect symmetry (upper triangular)

            long long max_j = max_p / i;
            if (max_j > N) max_j = N;

            // Loop for checking products
            for (long long j = min_j; j <= max_j; j++) {
                long long prod = (long long)i * j;
                if (prod >= min_p && prod <= max_p) {
                    // Map product to bit position
                    long long offset = prod - min_p;
                    long long byte = offset / 8;
                    int bit = offset % 8;
                    segment[byte] |= (1 << bit);
                }
            }
        }

        // Synchronize before reduction
        MPI_Barrier(MPI_COMM_WORLD);

        unsigned char *merged = NULL;
        if (rank == 0) {
            merged = (unsigned char *)calloc((SEGMENT_SIZE / 8) + 1, sizeof(unsigned char));
            if (!merged) {
                MPI_Abort(MPI_COMM_WORLD, 3);
                return -1;
            }
        }

        // Reduce bit arrays using bitwise OR
        MPI_Reduce(segment, merged, (SEGMENT_SIZE / 8) + 1, MPI_UNSIGNED_CHAR, MPI_BOR, 0,
MPI_COMM_WORLD);
```

```
        if (rank == 0) {
            long long segment_count = 0;
            for (long long i = 0; i < (SEGMENT_SIZE / 8) + 1; i++) {
                unsigned char byte = merged[i];
                while (byte) {
                    if (byte & 1) segment_count++;
                    byte >>= 1;
                }
            }

            total_unique += segment_count;
            free(merged);
        }
    }

    free(segment);

    // Broadcast the final result to all processes
    MPI_Bcast(&total_unique, 1, MPI_LONG_LONG_INT, 0, MPI_COMM_WORLD);

    return total_unique;
}

int main(int argc, char **argv) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (argc < 2) {
        if (rank == 0) {
            printf("Usage: %s <N1> [N2 N3 ...]\n", argv[0]);
        }
        MPI_Finalize();
        return 1;
    }

    if (rank == 0) {
        printf("Verifying known results (sequential):\n");
```

```
        int m5 = compute_M_N_sequential(5);
        printf("M(5) = %d (Expected: 14)\n", m5);
        int m10 = compute_M_N_sequential(10);
        printf("M(10) = %d (Expected: 42)\n\n", m10);
    }

    for (int i = 1; i < argc; i++) {
        int N = atoi(argv[i]);
        if (N <= 0) {
            if (rank == 0) {
                printf("WARNING: Skipping invalid N value: %s\n", argv[i]);
            }
            continue;
        }

        if (rank == 0) {
            printf("\n========================================\n");
            printf("Computing M(%d) using %d processes...\n", N, size);
            printf("========================================\n");
        }

        double t_start = MPI_Wtime();

        long long result;

        result = compute_M_N_block_advanced(N);

        double t_end = MPI_Wtime();

        if (rank == 0) {
            printf("\nRESULT: M(%d) = %lld\n", N, result);
            printf("Time taken: %.6f seconds\n", t_end - t_start);
            printf("========================================\n\n");
        }
    }

    MPI_Finalize();
    return 0;
}
```

bit_array_cannon.c

```c
/**
 * Title: Parallel Processing Assignment - Compute Unique Products M(N)
 * ---------------------------------------------------------------------------------------
 * Description: This program computes the number of unique products (i * j) for 1 <= i <= j <= N.
 *              Using advanced bit array segmentation in conjunction with Cannon’s Algorithm.
 */

// Sequential implementation for verification
int compute_M_N_sequential(int N) {
    bool *exists = (bool *)calloc((N * N) + 1, sizeof(bool));
    if (!exists) {
        printf("Failed in sequential\n");
        return -1;
    }

    // Nested loop from 1 <= i <= j <= N check i*j
    int count = 0;
    for (int i = 1; i <= N; i++) {
        for (int j = i; j <= N; j++) {
            int prod = i * j;
            if (!exists[prod]) {
                exists[prod] = true;
                count++;
            }
        }
    }
    free(exists);
    return count;
}

// Advanced parallel version using Cannon's algorithm with segmented bit arrays
long long compute_M_N_cannon_advanced(int N) {
    // Set up MPI variables
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Check if number of processes is a perfect square
    int grid_size = (int)sqrt(size);
```

```
    if (grid_size * grid_size != size) {
        if (rank == 0) {
            printf("Error: Number of processes must be a perfect square for Cannon's
algorithm\n");
        }
        MPI_Abort(MPI_COMM_WORLD, 1);
        return -1;
    }

    // Create 2D grid communicator
    int dims[2] = {grid_size, grid_size};
    int periods[2] = {1, 1}; // Periodic for cyclic shifts
    int reorder = 0;
    MPI_Comm grid_comm;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &grid_comm);

    // Get coordinates in the grid
    int coords[2];
    MPI_Cart_coords(grid_comm, rank, 2, coords);
    int row = coords[0];
    int col = coords[1];

    // Calculate local block boundaries using 2D decomposition
    int block_size = (N + grid_size - 1) / grid_size; // Ceiling division
    int i_start = row * block_size + 1;
    int i_end = (row + 1) * block_size;
    if (i_end > N) i_end = N;

    int j_start = col * block_size + 1;
    int j_end = (col + 1) * block_size;
    if (j_end > N) j_end = N;

    // Setup for segmented bit array processing
    long long max_prod = (long long)N * N;
    const long long SEGMENT_SIZE = 100000000LL; // 100M elements per segment
    long long num_segments = (max_prod / SEGMENT_SIZE) + 1;

    // Allocate memory for one segment's bit array
    unsigned char *segment = (unsigned char *)calloc((SEGMENT_SIZE / 8) + 1, sizeof(unsigned
char));
```

```
    if (!segment) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return -1;
    }

    long long total_unique = 0;

    // Process each segment of the product space
    for (long long s = 0; s < num_segments; s++) {
        // Clear segment buffer
        memset(segment, 0, (SEGMENT_SIZE / 8) + 1);

        // Define range for this segment
        long long min_p = s * SEGMENT_SIZE + 1;
        long long max_p = (s + 1) * SEGMENT_SIZE;
        if (max_p > max_prod) max_p = max_prod;

        // Process local block (only upper triangular part i <= j)
        for (int i = i_start; i <= i_end; i++) {
            // Determine j range for this segment
            long long min_j = min_p / i;
            if (min_j < i) min_j = i;  // respect i <= j for upper triangular
            if (min_j < j_start) min_j = j_start;  // respect block boundaries

            long long max_j = max_p / i;
            if (max_j > N) max_j = N;
            if (max_j > j_end) max_j = j_end;  // respect block boundaries

            // Process all valid products in this segment for current i
            for (long long j = min_j; j <= max_j; j++) {
                long long prod = (long long)i * j;
                if (prod >= min_p && prod <= max_p) {
                    // Map product to bit position
                    long long offset = prod - min_p;
                    long long byte = offset / 8;
                    int bit = offset % 8;
                    segment[byte] |= (1 << bit);
                }
            }
        }
```

```
// Create row and column communicators for Cannon's shifts
MPI_Comm row_comm, col_comm;
MPI_Comm_split(grid_comm, row, col, &row_comm);
MPI_Comm_split(grid_comm, col, row, &col_comm);

// Perform grid_size-1 circular shifts (Cannon's algorithm pattern)
for (int shift = 1; shift < grid_size; shift++) {
    // Compute shift source and destination
    int src_col = (col + 1) % grid_size;
    int dest_col = (col + grid_size - 1) % grid_size;

    // Share block boundaries for the next iteration
    int send_bounds[4] = {i_start, i_end, j_start, j_end};
    int recv_bounds[4];

    MPI_Sendrecv(send_bounds, 4, MPI_INT, dest_col, 0,
                 recv_bounds, 4, MPI_INT, src_col, 0,
                 row_comm, MPI_STATUS_IGNORE);

    // Update local boundaries for the next block
    int new_i_start = recv_bounds[0];
    int new_i_end = recv_bounds[1];
    int new_j_start = recv_bounds[2];
    int new_j_end = recv_bounds[3];

    // Process new block
    for (int i = new_i_start; i <= new_i_end; i++) {
        // Determine j range for this segment
        long long min_j = min_p / i;
        if (min_j < i) min_j = i;  // respect i <= j
        if (min_j < new_j_start) min_j = new_j_start;  // respect block

        long long max_j = max_p / i;
        if (max_j > N) max_j = N;
        if (max_j > new_j_end) max_j = new_j_end;  // respect block

        // Process all valid products in this segment for current i
        for (long long j = min_j; j <= max_j; j++) {
            long long prod = (long long)i * j;
```

```
                    if (prod >= min_p && prod <= max_p) {
                        // Map product to bit position
                        long long offset = prod - min_p;
                        int byte = offset / 8;
                        int bit = offset % 8;
                        segment[byte] |= (1 << bit);
                    }
                }
            }

            // Barrier to synchronize before next shift
            MPI_Barrier(grid_comm);
        }

        // Free communicators
        MPI_Comm_free(&row_comm);
        MPI_Comm_free(&col_comm);

        // Merge segment results across all processes
        unsigned char *merged_segment = NULL;
        if (rank == 0) {
            merged_segment = (unsigned char *)calloc((SEGMENT_SIZE / 8) + 1, sizeof(unsigned
char));
            if (!merged_segment) {
                MPI_Abort(MPI_COMM_WORLD, 3);
                return -1;
            }
        }

        // Reduce bit arrays using bitwise OR
        MPI_Reduce(segment, merged_segment, (SEGMENT_SIZE / 8) + 1, MPI_UNSIGNED_CHAR, MPI_BOR, 0,
MPI_COMM_WORLD);

        // Count unique products in this segment
        if (rank == 0) {
            long long segment_count = 0;
            for (long long i = 0; i < (SEGMENT_SIZE / 8) + 1; i++) {
                unsigned char byte = merged_segment[i];
                while (byte) {
                    if (byte & 1) segment_count++;
```

```
                    byte >>= 1;
                }
            }

            total_unique += segment_count;
            free(merged_segment);
        }
    }

    free(segment);
    MPI_Comm_free(&grid_comm);

    // Broadcast the final result to all processes
    MPI_Bcast(&total_unique, 1, MPI_LONG_LONG_INT, 0, MPI_COMM_WORLD);

    return total_unique;
}

int main(int argc, char **argv) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (argc < 2) {
        if (rank == 0) {
            printf("Usage: %s <N1> [N2 N3 ...]\n", argv[0]);
        }
        MPI_Finalize();
        return 1;
    }

    // Check if number of processes is a perfect square
    int grid_size = (int)sqrt(size);
    if (grid_size * grid_size != size) {
        if (rank == 0) {
            printf("Error: Number of processes must be a perfect square for Cannon's
algorithm\n");
            printf("Current number of processes: %d\n", size);
            printf("Try running with %d or %d processes\n",
```

```
                    (int)pow((int)sqrt(size), 2), (int)pow((int)sqrt(size) + 1, 2));
        }
        MPI_Finalize();
        return 1;
    }

    if (rank == 0 && size <= 16) {  // Only run sequential verification with small process counts
        printf("Verifying known results (sequential):\n");
        int m5 = compute_M_N_sequential(5);
        printf("M(5) = %d (Expected: 14)\n", m5);
        int m10 = compute_M_N_sequential(10);
        printf("M(10) = %d (Expected: 42)\n\n", m10);
    }

    for (int i = 1; i < argc; i++) {
        int N = atoi(argv[i]);
        if (N <= 0) {
            if (rank == 0) {
                printf("WARNING: Skipping invalid N value: %s\n", argv[i]);
            }
            continue;
        }

        if (rank == 0) {
            printf("\n========================================\n");
            printf("Computing M(%d) using %d processes (%dx%d grid)...\n", N, size, grid_size,
grid_size);
            printf("========================================\n");
        }

        double t_start = MPI_Wtime();

        long long result = compute_M_N_cannon_advanced(N);

        double t_end = MPI_Wtime();

        if (rank == 0) {
            printf("\nRESULT: M(%d) = %lld\n", N, result);
            printf("Time taken: %.6f seconds\n", t_end - t_start);
            printf("========================================\n\n");
```

```
        }
    }

    MPI_Finalize();
    return 0;
}
```

Optimized_Cyclic.c

```
/**
 * Title: Parallel Processing Assignment - Compute Unique Products M(N)
 * ------------------------------------------------------------------------------------------
 * Description: This program computes the number of unique products (i * j) for 1 <= i <= j <= N.
 *              Using first factor memory optimization and cyclic load balancing.
 */

// First factor function
bool is_first_factor(int i, int j, int N) {
    // For a product i*j, this function checks if (i,j) is the
    // first valid representation - meaning i is the smallest factor
    // of the product that's within our range [1,N]

    long long prod = (long long)i * j;

    // Check all smaller potential factors
    for (int k = 1; k < i; k++) {
        if (prod % k == 0) {
            // If k is a factor, we need to check if the other factor is in range
            long long other = prod / k;
            if (other <= N && other >= k) {
                // Found a smaller valid factor - this is not the first factor
                return false;
            }
        }
    }

    // No smaller valid factor pair found - this is first factor
    return true;
}
```

```
// Sequential implementation for verification
int compute_M_N_sequential(int N) {
    bool *exists = (bool *)calloc((N * N) + 1, sizeof(bool));
    if (!exists) {
        printf("Failed in sequential\n");
        return -1;
    }

    int count = 0;

    for (int i = 1; i <= N; i++) {
        for (int j = i; j <= N; j++) {
            int prod = i * j;
            if (!exists[prod]) {
                exists[prod] = true;
                count++;
            }
        }
    }
    free(exists);
    return count;
}

// Optimized parallel version
int compute_M_N_optimized(int N) {
    // Set up MPI variables
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int local_count = 0;

    // Each process handles rows based on rank
    for (int i = 1 + rank; i <= N; i += size) {
        for (int j = i; j <= N; j++) {
            // Check if this is the first factor representation for this product
            if (is_first_factor(i, j, N)) {
                local_count++;
            }
        }
```

```
    }

    // Reduce to get total count
    int total_count = 0;
    MPI_Reduce(&local_count, &total_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Broadcast the result to all processes
    MPI_Bcast(&total_count, 1, MPI_INT, 0, MPI_COMM_WORLD);

    return total_count;
}

int main(int argc, char **argv) {
    // Set up MPI variables
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (argc < 2) {
        if (rank == 0) {
            printf("Usage: %s <N1> [N2 N3 ...]\n", argv[0]);
        }
        MPI_Finalize();
        return 1;
    }

    if (rank == 0) {
        printf("Verifying known results (sequential):\n");
        int m5 = compute_M_N_sequential(5);
        printf("M(5) = %d (Expected: 14)\n", m5);
        int m10 = compute_M_N_sequential(10);
        printf("M(10) = %d (Expected: 42)\n\n", m10);
    }

    for (int i = 1; i < argc; i++) {
        int N = atoi(argv[i]);
        if (N <= 0) {
            if (rank == 0) {
                printf("WARNING: Skipping invalid N value: %s\n", argv[i]);
```

```
            }
            continue;
        }

        if (rank == 0) {
            printf("\n========================================\n");
            printf("Computing M(%d) using %d processes...\n", N, size);
            printf("========================================\n");
        }

        double t_start = MPI_Wtime();

        int result;
        result = compute_M_N_optimized(N);

        double t_end = MPI_Wtime();

        if (rank == 0) {
            printf("\nRESULT: M(%d) = %d\n", N, result);
            printf("Time taken: %.6f seconds\n", t_end - t_start);
            printf("========================================\n\n");
        }
    }

    MPI_Finalize();
    return 0;
}
```

Optimized_Cannon.c

```
/**
 * Title: Parallel Processing Assignment - Compute Unique Products M(N)
 * ------------------------------------------------------------------------------------------
 * Description: This program computes the number of unique products (i * j) for 1 <= i <= j <= N.
 *              Using first factor memory optimization in conjunction with Cannon's Algorithm.
 */

// First factor function
bool is_first_factor(int i, int j, int N) {
    // For a product i*j, this function checks if (i,j) is the
```

```
    // first valid representation - meaning i is the smallest factor
    // of the product that's within our range [1,N]

    long long prod = (long long)i * j;

    // Check all smaller potential factors
    for (int k = 1; k < i; k++) {
        if (prod % k == 0) {
            // If k is a factor, we need to check if the other factor is in range
            long long other = prod / k;
            if (other <= N && other >= k) {
                // Found a smaller valid factor - this is not the first factor
                return false;
            }
        }
    }

    // No smaller valid factor pair found - this is first factor
    return true;
}


// Sequential implementation for verification
int compute_M_N_sequential(int N) {
    bool *exists = (bool *)calloc((N * N) + 1, sizeof(bool));
    if (!exists) {
        printf("Failed in sequential\n");
        return -1;
    }

    int count = 0;

    for (int i = 1; i <= N; i++) {
        for (int j = i; j <= N; j++) {
            int prod = i * j;
            if (!exists[prod]) {
                exists[prod] = true;
                count++;
            }
        }
```

```
    }
    free(exists);
    return count;
}

// Cannon's algorithm implementation for M(N)
int compute_M_N_cannon(int N) {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Check if number of processes is a perfect square
    int grid_size = (int)sqrt(size);
    if (grid_size * grid_size != size) {
        if (rank == 0) {
            printf("Error: Number of processes must be a perfect square for Cannon's
algorithm\n");
        }
        MPI_Abort(MPI_COMM_WORLD, 1);
        return -1;
    }

    // Create 2D grid communicator
    int dims[2] = {grid_size, grid_size};
    int periods[2] = {1, 1}; // Periodic in both dimensions for cyclic shifts
    int reorder = 0;
    MPI_Comm grid_comm;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &grid_comm);

    // Get coordinates in the grid
    int coords[2];
    MPI_Cart_coords(grid_comm, rank, 2, coords);
    int row = coords[0];
    int col = coords[1];

    // Calculate local block boundaries
    int block_size = (N + grid_size - 1) / grid_size; // Ceiling division
    int i_start = row * block_size + 1;
    int i_end = (row + 1) * block_size;
    if (i_end > N) i_end = N;
```

```
int j_start = col * block_size + 1;
int j_end = (col + 1) * block_size;
if (j_end > N) j_end = N;

// Create a structure to track which (i,j) pairs we've already processed
// to avoid double counting during shifts
bool **processed = (bool **)malloc((N + 1) * sizeof(bool *));
for (int i = 0; i <= N; i++) {
    processed[i] = (bool *)calloc(N + 1, sizeof(bool));
}

// Count unique values in local block (upper triangular portion only)
int local_count = 0;
for (int i = i_start; i <= i_end; i++) {
    for (int j = j_start; j <= j_end; j++) {
        if (i <= j) { // Consider only upper triangular
            // Mark this pair as processed
            processed[i][j] = true;

            // Use first factor check
            if (is_first_factor(i, j, N)) {
                local_count++;
            }
        }
    }
}

// Define row and column communicators for shifts
MPI_Comm row_comm, col_comm;
MPI_Comm_split(grid_comm, row, col, &row_comm);
MPI_Comm_split(grid_comm, col, row, &col_comm);

// Initialize buffer for receiving blocks
int recv_i_start, recv_i_end, recv_j_start, recv_j_end;

// Perform grid_size-1 shifts
for (int shift = 1; shift < grid_size; shift++) {
    // Shift row blocks left
    int src_col = (col + 1) % grid_size;
```

```
    int dest_col = (col + grid_size - 1) % grid_size;

    // Pack and send current block boundaries
    int send_bounds[4] = {i_start, i_end, j_start, j_end};
    int recv_bounds[4];

    MPI_Sendrecv(send_bounds, 4, MPI_INT, dest_col, 0,
                 recv_bounds, 4, MPI_INT, src_col, 0,
                 row_comm, MPI_STATUS_IGNORE);

    recv_i_start = recv_bounds[0];
    recv_i_end = recv_bounds[1];
    recv_j_start = recv_bounds[2];
    recv_j_end = recv_bounds[3];

    // Process received block, but only if we haven't processed these pairs already
    for (int i = recv_i_start; i <= recv_i_end; i++) {
        for (int j = recv_j_start; j <= recv_j_end; j++) {
            if (i <= j && !processed[i][j]) { // Upper triangular and not processed yet
                // Mark as processed to avoid future double counting
                processed[i][j] = true;

                // Use corrected Cannonical pair check
                if (is_cannonical_pair(i, j, N)) {
                    local_count++;
                }
            }
        }
    }

    // Synchronize before next shift
    MPI_Barrier(grid_comm);
}

// Free processed array
for (int i = 0; i <= N; i++) {
    free(processed[i]);
}
free(processed);
```

```
    // Reduce all local counts to get the total
    int total_count = 0;
    MPI_Allreduce(&local_count, &total_count, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    // Free communicators
    MPI_Comm_free(&row_comm);
    MPI_Comm_free(&col_comm);
    MPI_Comm_free(&grid_comm);

    return total_count;
}

int main(int argc, char **argv) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (argc < 2) {
        if (rank == 0) {
            printf("Usage: %s <N1> [N2 N3 ...]\n", argv[0]);
        }
        MPI_Finalize();
        return 1;
    }

    if (rank == 0) {
        printf("Verifying known results (sequential):\n");
        int m5 = compute_M_N_sequential(5);
        printf("M(5) = %d (Expected: 14)\n", m5);
        int m10 = compute_M_N_sequential(10);
        printf("M(10) = %d (Expected: 42)\n\n", m10);
    }

    for (int i = 1; i < argc; i++) {
        int N = atoi(argv[i]);
        if (N <= 0) {
            if (rank == 0) {
                printf("WARNING: Skipping invalid N value: %s\n", argv[i]);
            }
```

```
            continue;
        }

        if (rank == 0) {
            printf("\n========================================\n");
            printf("Computing M(%d) using %d processes...\n", N, size);
            printf("========================================\n");
        }

        double t_start = MPI_Wtime();

        int result;
        int grid_size = (int)sqrt(size);

        if (grid_size * grid_size == size && N <= 100000) {
            // Use Cannon's algorithm if number of processes is a perfect square
            result = compute_M_N_cannon(N);
        } else {
            // Fall back to optimized approach otherwise
            if (rank == 0) {
                printf("Number of processes must be a perfect square for Cannon's algorithm\n");
            }
            MPI_Finalize();
            return 1;
        }

        double t_end = MPI_Wtime();

        result = result/2;

        if (rank == 0) {
            printf("\nRESULT: M(%d) = %d\n", N, result);
            printf("Time taken: %.6f seconds\n", t_end - t_start);
            printf("========================================\n\n");
        }
    }

    MPI_Finalize();
    return 0;
}
```

Output Code Results:

```
[lcl_uotwlus1104@teach-login01 term project]$ mpirun -np 4 ./canon-bit-array 5 10 20 40 80 160 320 640 1000 2000 8000 16000 32000 40000 50000 60000 70000 80000 90000 100000
Verifying known results (sequential):
M(5) = 14 (Expected: 14)
M(10) = 42 (Expected: 42)


========================================
Computing M(5) using 4 processes (2x2 grid)...
========================================

RESULT: M(5) = 14
Time taken: 2.005809 seconds
========================================


========================================
Computing M(10) using 4 processes (2x2 grid)...
========================================

RESULT: M(10) = 42
Time taken: 2.322668 seconds
========================================


========================================
Computing M(20) using 4 processes (2x2 grid)...
========================================

RESULT: M(20) = 152
Time taken: 2.342455 seconds
========================================


========================================
Computing M(40) using 4 processes (2x2 grid)...
========================================

RESULT: M(40) = 517
Time taken: 2.232938 seconds
========================================


========================================
Computing M(80) using 4 processes (2x2 grid)...
========================================

RESULT: M(80) = 1939
Time taken: 2.038650 seconds
========================================
```

```
Computing M(80) using 4 processes (2x2 grid)...
========================================

RESULT: M(80) = 1939
Time taken: 2.038650 seconds
========================================


========================================
Computing M(160) using 4 processes (2x2 grid)...
========================================

RESULT: M(160) = 7174
Time taken: 2.236501 seconds
========================================


========================================
Computing M(320) using 4 processes (2x2 grid)...
========================================

RESULT: M(320) = 27354
Time taken: 2.231185 seconds
========================================


========================================
Computing M(640) using 4 processes (2x2 grid)...
========================================

RESULT: M(640) = 103966
Time taken: 2.241483 seconds
========================================


========================================
Computing M(1000) using 4 processes (2x2 grid)...
========================================

RESULT: M(1000) = 248083
Time taken: 2.408616 seconds
========================================


========================================
Computing M(2000) using 4 processes (2x2 grid)...
========================================

RESULT: M(2000) = 959759
Time taken: 3.357899 seconds
========================================
```

```
Computing M(8000) using 4 processes (2x2 grid)...
========================================

RESULT: M(8000) = 14509549
Time taken: 0.493007 seconds
========================================


========================================
Computing M(16000) using 4 processes (2x2 grid)...
========================================

RESULT: M(16000) = 56705617
Time taken: 2.293561 seconds
========================================


========================================
Computing M(32000) using 4 processes (2x2 grid)...
========================================

RESULT: M(32000) = 221824366
Time taken: 8.631922 seconds
========================================


========================================
Computing M(40000) using 4 processes (2x2 grid)...
========================================

RESULT: M(40000) = 344462009
Time taken: 13.606368 seconds
========================================


========================================
Computing M(50000) using 4 processes (2x2 grid)...
========================================

RESULT: M(50000) = 534772334
Time taken: 21.451990 seconds
========================================


========================================
Computing M(60000) using 4 processes (2x2 grid)...
========================================
```

```
RESULT: M(50000) = 534772334
Time taken: 21.451990 seconds
========================================


========================================
Computing M(60000) using 4 processes (2x2 grid)...
========================================

RESULT: M(60000) = 766265795
Time taken: 30.991873 seconds
========================================


========================================
Computing M(70000) using 4 processes (2x2 grid)...
========================================

RESULT: M(70000) = 1038159781
Time taken: 42.428553 seconds
========================================


========================================
Computing M(80000) using 4 processes (2x2 grid)...
========================================

RESULT: M(80000) = 1351433197
Time taken: 55.367532 seconds
========================================


========================================
Computing M(90000) using 4 processes (2x2 grid)...
========================================

RESULT: M(90000) = 1704858198
Time taken: 70.188177 seconds
========================================


========================================
Computing M(100000) using 4 processes (2x2 grid)...
========================================

RESULT: M(100000) = 2099198630
Time taken: 86.729071 seconds
========================================
```

```
[lcl_uotwlus1104@teach-login01 term project]$  mpirun -np 4 ./cyclic-bit-array 5 10 20 40 80 160 320 640 1000 2000 8000 16000 32000 40000 50000 60000 70000 80000 90000 100000
Verifying known results (sequential):
M(5) = 14 (Expected: 14)
M(10) = 42 (Expected: 42)


========================================
Computing M(5) using 4 processes...
========================================

RESULT: M(5) = 14
Time taken: 0.949693 seconds
========================================


========================================
Computing M(10) using 4 processes...
========================================

RESULT: M(10) = 42
Time taken: 0.949444 seconds
========================================


========================================
Computing M(20) using 4 processes...
========================================

RESULT: M(20) = 152
Time taken: 1.033053 seconds
========================================


========================================
Computing M(40) using 4 processes...
========================================

RESULT: M(40) = 517
Time taken: 0.950310 seconds
========================================


========================================
Computing M(80) using 4 processes...
========================================

RESULT: M(80) = 1939
Time taken: 1.038913 seconds
========================================
```

```
========================================
Computing M(160) using 4 processes...
========================================

RESULT: M(160) = 7174
Time taken: 0.047229 seconds
========================================


========================================
Computing M(320) using 4 processes...
========================================

RESULT: M(320) = 27354
Time taken: 0.047700 seconds
========================================


========================================
Computing M(640) using 4 processes...
========================================

RESULT: M(640) = 103966
Time taken: 0.049157 seconds
========================================


========================================
Computing M(1000) using 4 processes...
========================================

RESULT: M(1000) = 248083
Time taken: 0.051908 seconds
========================================


========================================
Computing M(2000) using 4 processes...
========================================

RESULT: M(2000) = 959759
Time taken: 0.065504 seconds
========================================


========================================
Computing M(8000) using 4 processes...
========================================
```

```
========================================
Computing M(8000) using 4 processes...
========================================

RESULT: M(8000) = 14509549
Time taken: 0.347372 seconds
========================================


========================================
Computing M(16000) using 4 processes...
========================================

RESULT: M(16000) = 56705617
Time taken: 1.368232 seconds
========================================


========================================
Computing M(32000) using 4 processes...
========================================

RESULT: M(32000) = 221824366
Time taken: 5.440158 seconds
========================================


========================================
Computing M(40000) using 4 processes...
========================================

RESULT: M(40000) = 344462009
Time taken: 8.489121 seconds
========================================


========================================
Computing M(50000) using 4 processes...
========================================

RESULT: M(50000) = 534772334
Time taken: 13.234685 seconds
========================================


========================================
Computing M(60000) using 4 processes...
========================================
```

```
========================================
Computing M(70000) using 4 processes...
========================================

RESULT: M(70000) = 1038159781
Time taken: 25.837963 seconds
========================================


========================================
Computing M(80000) using 4 processes...
========================================

RESULT: M(80000) = 1351433197
Time taken: 33.793175 seconds
========================================


========================================
Computing M(90000) using 4 processes...
========================================

RESULT: M(90000) = 1704858198
Time taken: 42.648385 seconds
========================================


========================================
Computing M(100000) using 4 processes...
========================================

RESULT: M(100000) = 2099198630
Time taken: 52.555800 seconds
========================================
```

```
[lcl_uotwlus1104@teach-login01 term project]$  mpirun -np 4 ./block-bit-array 5 10 20 40 80 160 320 640 1000 2000 8000 16000 32000 40000 50000 60000 70000 80000 90000 100000
[mii] Please select a module to run mpirun:
```

```
Verifying known results (sequential):
M(5) = 14 (Expected: 14)
M(10) = 42 (Expected: 42)


========================================
Computing M(5) using 4 processes with block distribution...
========================================
Process 0 handles rows 1 to 2 (2 rows)

RESULT: M(5) = 14
Time taken: 0.048033 seconds
========================================


========================================
Computing M(10) using 4 processes with block distribution...
========================================
Process 0 handles rows 1 to 3 (3 rows)

RESULT: M(10) = 42
Time taken: 0.045388 seconds
========================================


========================================
Computing M(20) using 4 processes with block distribution...
========================================
Process 0 handles rows 1 to 5 (5 rows)

RESULT: M(20) = 152
Time taken: 0.046516 seconds
========================================


========================================
Computing M(40) using 4 processes with block distribution...
========================================
Process 0 handles rows 1 to 10 (10 rows)

RESULT: M(40) = 517
Time taken: 0.046399 seconds
========================================


========================================
Computing M(80) using 4 processes with block distribution...
========================================
```

```
RESULT: M(40) = 517
Time taken: 0.046399 seconds
========================================


========================================
Computing M(80) using 4 processes with block distribution...
========================================
Process 0 handles rows 1 to 20 (20 rows)

RESULT: M(80) = 1939
Time taken: 0.046387 seconds
========================================


========================================
Computing M(160) using 4 processes with block distribution...
========================================
Process 0 handles rows 1 to 40 (40 rows)

RESULT: M(160) = 7174
Time taken: 0.046423 seconds
========================================


========================================
Computing M(320) using 4 processes with block distribution...
========================================
Process 0 handles rows 1 to 80 (80 rows)

RESULT: M(320) = 27354
Time taken: 0.046908 seconds
========================================


========================================
Computing M(640) using 4 processes with block distribution...
========================================
Process 0 handles rows 1 to 160 (160 rows)

RESULT: M(640) = 103966
Time taken: 0.048694 seconds
========================================


========================================
Computing M(1000) using 4 processes with block distribution...
========================================
```

```
Process 0 handles rows 1 to 250 (250 rows)

RESULT: M(1000) = 248083
Time taken: 0.051710 seconds
========================================


========================================
Computing M(2000) using 4 processes with block distribution...
========================================
Process 0 handles rows 1 to 500 (500 rows)

RESULT: M(2000) = 959759
Time taken: 0.066725 seconds
========================================


========================================
Computing M(8000) using 4 processes with block distribution...
========================================
Process 0 handles rows 1 to 2000 (2000 rows)

RESULT: M(8000) = 14509549
Time taken: 0.376975 seconds
========================================


========================================
Computing M(16000) using 4 processes with block distribution...
========================================
Process 0 handles rows 1 to 4000 (4000 rows)

RESULT: M(16000) = 56705617
Time taken: 1.534098 seconds
========================================


========================================
Computing M(32000) using 4 processes with block distribution...
========================================
Process 0 handles rows 1 to 8000 (8000 rows)

RESULT: M(32000) = 221824366
Time taken: 6.141975 seconds
========================================


========================================
```

```
Time taken: 15.081690 seconds
========================================


========================================
Computing M(60000) using 4 processes with block distribution...
========================================
Process 0 handles rows 1 to 15000 (15000 rows)

RESULT: M(60000) = 766265795
Time taken: 21.743348 seconds
========================================


========================================
Computing M(70000) using 4 processes with block distribution...
========================================
Process 0 handles rows 1 to 17500 (17500 rows)

RESULT: M(70000) = 1038159781
Time taken: 29.653188 seconds
========================================


========================================
Computing M(80000) using 4 processes with block distribution...
========================================
Process 0 handles rows 1 to 20000 (20000 rows)

RESULT: M(80000) = 1351433197
Time taken: 38.787977 seconds
========================================


========================================
Computing M(90000) using 4 processes with block distribution...
========================================
Process 0 handles rows 1 to 22500 (22500 rows)

RESULT: M(90000) = 1704858198
Time taken: 49.130687 seconds
========================================


========================================
Computing M(100000) using 4 processes with block distribution...
========================================
Process 0 handles rows 1 to 25000 (25000 rows)
```

Benchmarking:

Performance benchmark for N = 5

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon
1	0.000005	0.025480	0.025230	0.028297	0.000005	0.000120
4	0.000040	0.046544	0.048033	0.049563	0.000031	0.000160
9	0.000220	0.055213	0.055055	0.055424	0.000032	0.000379
16	0.000210	0.058085	0.057188	0.058245	0.000090	0.000487
25	0.000281	0.064180	0.063127	0.085090	0.025941	0.389498

Performance benchmark for N = 10

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon
1	0.000002	0.021001	0.020808	0.022700	0.000005	0.000024
4	0.000025	0.045496	0.045388	0.046186	0.000003	0.000078
9	0.000046	0.053275	0.054173	0.054339	0.000003	0.000100
16	0.000058	0.056614	0.056822	0.057961	0.000003	0.000113
25	0.000076	0.062076	0.061133	0.063753	0.051946	0.374899

Performance benchmark for N = 20

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon

1	0.000004	0.020952	0.021023	0.020032	0.000028	0.000047
4	0.000009	0.045682	0.046516	0.047858	0.000009	0.000077
9	0.000021	0.054833	0.053942	0.054350	0.000005	0.000113
16	0.000018	0.057629	0.056972	0.057371	0.000004	0.000116
25	0.000028	0.062965	0.061605	0.063493	0.051985	0.288004

Performance benchmark for N = 40

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon
1	0.000013	0.020849	0.022125	0.020051	0.000186	0.000211
4	0.000018	0.045937	0.046399	0.048276	0.000045	0.000379
9	0.000036	0.054588	0.053919	0.055243	0.000021	0.000224
16	0.000051	0.056907	0.056717	0.056940	0.000013	0.000188
25	0.000053	0.061958	0.062379	0.066770	0.051981	0.273750

Performance benchmark for N = 80

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon
1	0.000055	0.020956	0.021158	0.020547	0.001191	0.001235
4	0.000090	0.048596	0.046387	0.047998	0.000319	0.001422
9	0.000085	0.054822	0.053850	0.055000	0.000142	0.000964

16	0.000127	0.056294	0.055819	0.056944	0.000077	0.000707
25	0.000132	0.061890	0.061315	0.066120	0.051982	0.202889

Performance benchmark for N = 160

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon
1	0.000212	0.020993	0.021235	0.020252	0.007934	0.008015
4	0.000251	0.047229	0.046423	0.048368	0.002079	0.008036
9	0.000294	0.054684	0.054053	0.054863	0.000990	0.005732
16	0.000280	0.056106	0.055964	0.057182	0.000575	0.004040
25	0.000316	0.061720	0.064253	0.067058	0.051980	0.207927

Performance benchmark for N = 320

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon
1	0.000759	0.021463	0.021105	0.020864	0.056962	0.122175
4	0.000887	0.047700	0.046908	0.048792	0.014328	0.057067
9	0.000999	0.054965	0.054867	0.055359	0.006525	0.038873
16	0.001036	0.057251	0.055913	0.057039	0.003740	0.027479
25	0.001111	0.062416	0.064166	0.067060	0.051978	0.207871

Performance benchmark for N = 640

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon
1	0.002907	0.022497	0.022620	0.022010	0.421515	0.849766
4	0.000887	0.049157	0.048694	0.050724	0.103540	0.427194
9	0.003683	0.056359	0.055746	0.057503	0.046701	0.287749
16	0.003717	0.057632	0.057385	0.059067	0.026160	0.204131
25	0.003813	0.063307	0.065304	0.068621	0.079981	0.383870

Performance benchmark for N = 1000

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon
1	0.007351	0.024891	0.024704	0.025393	1.545316	3.059727
4	0.007249	0.051908	0.051710	0.054331	0.376487	1.583326
9	0.008180	0.058844	0.058544	0.060911	0.168908	1.063261
16	0.008197	0.060397	0.059617	0.061755	0.094568	0.757075
25	0.008477	0.066083	0.067825	0.072130	0.147589	0.738922

Performance benchmark for N = 2000

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon

1	0.027748	0.036384	0.036403	0.036016	11.830161	23.451408
4	0.028115	0.065504	0.066725	0.072387	2.907338	12.264726
9	0.031134	0.072462	0.07133	0.078214	1.282809	8.184642
16	0.030612	0.072946	0.072597	0.078276	0.726995	5.866201
25	0.031861	0.078022	0.078894	0.085580	0.974470	4.852891

Performance benchmark for N = 8000

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon
1	0.504271	0.275613	0.303291	0.283631	1348.423483	1417.156170
4	0.487673	0.347372	0.376975	0.493007	178.898074	758.687427
9	0.478143	0.324421	0.336872	0.442236	78.607553	503.968842
16	0.472326	0.316690	0.321481	0.407310	44.740368	362.913111
25	0.492784	0.316568	0.318291	0.385985	39.882981	296.455811

Performance benchmark for N = 16,000

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon
1	2.704892	1.095530	1.118999	1.311387	DNF	DNF
4	2.111424	1.368232	1.534098	2.293561	1416.099513	DNF

9	1.969355	1.267211	1.315586	2.087108	621.847429	DNF
16	1.889844	1.239115	1.254280	2.153650	354.241536	2882.078682
25	1.974310	1.230493	1.239478	2.019938	303.418342	3147.422869

Performance benchmark for N = 32,000

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon
1	13.987126	4.512644	7.001417	5.560011	DNF	DNF
4	8.687078	5.440158	6.141975	8.631922	DNF	DNF
9	7.911520	5.070082	5.173290	9.983202	DNF	DNF
16	8.173827	4.828183	4.877488	11.366624	2812.784026	DNF
25	7.871757	4.786187	4.909442	12.230786	2951.085973	DNF

Performance benchmark for N = 40,000

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon
1	22.430775	6.968171	9.578330	8.469553	DNF	DNF
4	13.654003	8.489121	9.608803	13.606368	DNF	DNF
9	12.529540	7.798529	8.065470	15.882469	DNF	DNF

16	13.253679	7.437856	7.511777	18.221498	DNF	DNF
25	12.633755	7.396951	7.465304	20.457722	DNF	DNF

Performance benchmark for N = 50,000

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon
1	DNF	11.076311	17.318775	13.555085	DNF	DNF
4	DNF	13.234685	15.081690	21.451990	DNF	DNF
9	DNF	11.999548	12.576673	26.091551	DNF	DNF
16	DNF	11.499121	11.707929	29.428584	DNF	DNF
25	DNF	11.433776	11.430980	33.139867	DNF	DNF

Performance benchmark for N = 60,000

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon
1	DNF	15.961117	24.338447	22.102536	DNF	DNF
4	DNF	19.008338	21.743348	30.991873	DNF	DNF
9	DNF	17.232217	18.088442	37.668923	DNF	DNF
16	DNF	16.430711	16.833956	43.542032	DNF	DNF
25	DNF	16.339115	16.622011	48.515912	DNF	DNF

Performance benchmark for N = 70,000

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon
1	DNF	21.604208	34.300779	30.312118	DNF	DNF
4	DNF	25.837963	29.653188	42.428553	DNF	DNF
9	DNF	23.217536	24.608068	52.038788	DNF	DNF
16	DNF	22.237129	22.876571	60.178705	DNF	DNF
25	DNF	22.120452	22.583568	68.533848	DNF	DNF

Performance benchmark for N = 80,000

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon
1	DNF	28.152983	38.913345	40.043552	DNF	DNF
4	DNF	33.793175	38.787977	55.367532	DNF	DNF
9	DNF	30.370068	32.136284	69.795520	DNF	DNF
16	DNF	28.938943	29.859504	79.91889	DNF	DNF
25	DNF	28.894029	29.545236	90.702026	DNF	DNF

Performance benchmark for N = 90,000

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon

1	DNF	35.438466	54.701366	46.073131	DNF	DNF
4	DNF	42.648385	49.130687	70.188177	DNF	DNF
9	DNF	38.140889	40.632021	86.298726	DNF	DNF
16	DNF	36.515410	37.747332	102.003456	DNF	DNF
25	DNF	36.526583	37.386652	115.222522	DNF	DNF

Performance benchmark for N = 100,000

Processors	Parallel	Bit Array With Cyclic	Bit Array With Block	Bit Array With Cannon	Optimized with Cyclic	Optimized with Cannon
1	DNF	44.150632	62.416038	53.234932	DNF	DNF
4	DNF	52.555800	60.704886	86.729071	DNF	DNF
9	DNF	47.258143	50.165235	107.741419	DNF	DNF
16	DNF	44.959808	46.585156	127.212180	DNF	DNF
25	DNF	44.902574	46.189385	147.773271	DNF	DNF

Code for Making Graphs:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.colors import LogNorm
import matplotlib.ticker as ticker

# Set plot style
plt.style.use('ggplot')
sns.set_context("notebook", font_scale=1.2)
```

```
# Create a function to prepare the data
def prepare_benchmark_data():
    # Algorithm names
    algorithms = [
        "Parallel",
        "Bit Array With Cyclic",
        "Bit Array With Block",
        "Bit Array With Cannon",
        "Optimized with Cyclic",
        "Optimized with Cannon"
    ]

    # Problem sizes
    sizes = [5, 10, 20, 40, 80, 160, 320, 640, 1000, 2000, 8000, 16000, 32000, 40000, 50000,
60000, 70000, 80000, 90000, 100000]

    # Processor counts
    processors = [1, 4, 9, 16, 25]

    # Create empty DataFrame with "Algorithm", "N", "Processors", and "ExecutionTime" columns
    data = []

    # N = 5 data
    data.extend([
      {"N": 5, "Processors": 1, "Algorithm": "Parallel", "ExecutionTime": 0.000005},
      {"N": 5, "Processors": 1, "Algorithm": "Bit Array With Cyclic", "ExecutionTime": 0.025480},
      {"N": 5, "Processors": 1, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.025230},
      {"N": 5, "Processors": 1, "Algorithm": "Bit Array With Cannon", "ExecutionTime": 0.028297},
      {"N": 5, "Processors": 1, "Algorithm": "Optimized with Cyclic", "ExecutionTime": 0.000005},
      {"N": 5, "Processors": 1, "Algorithm": "Optimized with Cannon", "ExecutionTime": 0.000120},

      {"N": 5, "Processors": 4, "Algorithm": "Parallel", "ExecutionTime": 0.000040},
      {"N": 5, "Processors": 4, "Algorithm": "Bit Array With Cyclic", "ExecutionTime": 0.046544},
      {"N": 5, "Processors": 4, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.048033},
      {"N": 5, "Processors": 4, "Algorithm": "Bit Array With Cannon", "ExecutionTime": 0.049563},
      {"N": 5, "Processors": 4, "Algorithm": "Optimized with Cyclic", "ExecutionTime": 0.000031},
      {"N": 5, "Processors": 4, "Algorithm": "Optimized with Cannon", "ExecutionTime": 0.000160},

      {"N": 5, "Processors": 9, "Algorithm": "Parallel", "ExecutionTime": 0.000220},
      {"N": 5, "Processors": 9, "Algorithm": "Bit Array With Cyclic", "ExecutionTime": 0.055213},
```

```
    {"N": 5, "Processors": 9, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.055055},
    {"N": 5, "Processors": 9, "Algorithm": "Bit Array With Cannon", "ExecutionTime": 0.055424},
    {"N": 5, "Processors": 9, "Algorithm": "Optimized with Cyclic", "ExecutionTime": 0.000032},
    {"N": 5, "Processors": 9, "Algorithm": "Optimized with Cannon", "ExecutionTime": 0.000379},

    {"N": 5, "Processors": 16, "Algorithm": "Parallel", "ExecutionTime": 0.000210},
    {"N": 5, "Processors": 16, "Algorithm": "Bit Array With Cyclic", "ExecutionTime": 0.058085},
    {"N": 5, "Processors": 16, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.057188},
    {"N": 5, "Processors": 16, "Algorithm": "Bit Array With Cannon", "ExecutionTime": 0.058245},
    {"N": 5, "Processors": 16, "Algorithm": "Optimized with Cyclic", "ExecutionTime": 0.000090},
    {"N": 5, "Processors": 16, "Algorithm": "Optimized with Cannon", "ExecutionTime": 0.000487},

    {"N": 5, "Processors": 25, "Algorithm": "Parallel", "ExecutionTime": 0.000281},
    {"N": 5, "Processors": 25, "Algorithm": "Bit Array With Cyclic", "ExecutionTime": 0.064180},
    {"N": 5, "Processors": 25, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.063127},
    {"N": 5, "Processors": 25, "Algorithm": "Bit Array With Cannon", "ExecutionTime": 0.085090},
    {"N": 5, "Processors": 25, "Algorithm": "Optimized with Cyclic", "ExecutionTime": 0.025941},
    {"N": 5, "Processors": 25, "Algorithm": "Optimized with Cannon", "ExecutionTime": 0.389498},

  # N = 10 data
    {"N": 10, "Processors": 1, "Algorithm": "Parallel", "ExecutionTime": 0.000002},
    {"N": 10, "Processors": 1, "Algorithm": "Bit Array With Cyclic", "ExecutionTime": 0.021001},
    {"N": 10, "Processors": 1, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.020808},
    {"N": 10, "Processors": 1, "Algorithm": "Bit Array With Cannon", "ExecutionTime": 0.022700},
    {"N": 10, "Processors": 1, "Algorithm": "Optimized with Cyclic", "ExecutionTime": 0.000005},
    {"N": 10, "Processors": 1, "Algorithm": "Optimized with Cannon", "ExecutionTime": 0.000024},

    {"N": 10, "Processors": 4, "Algorithm": "Parallel", "ExecutionTime": 0.000025},
    {"N": 10, "Processors": 4, "Algorithm": "Bit Array With Cyclic", "ExecutionTime": 0.045496},
    {"N": 10, "Processors": 4, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.045388},
    {"N": 10, "Processors": 4, "Algorithm": "Bit Array With Cannon", "ExecutionTime": 0.046186},
    {"N": 10, "Processors": 4, "Algorithm": "Optimized with Cyclic", "ExecutionTime": 0.000003},
    {"N": 10, "Processors": 4, "Algorithm": "Optimized with Cannon", "ExecutionTime": 0.000078},

    {"N": 10, "Processors": 9, "Algorithm": "Parallel", "ExecutionTime": 0.000046},
    {"N": 10, "Processors": 9, "Algorithm": "Bit Array With Cyclic", "ExecutionTime": 0.053275},
    {"N": 10, "Processors": 9, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.054173},
    {"N": 10, "Processors": 9, "Algorithm": "Bit Array With Cannon", "ExecutionTime": 0.054339},
    {"N": 10, "Processors": 9, "Algorithm": "Optimized with Cyclic", "ExecutionTime": 0.000003},
    {"N": 10, "Processors": 9, "Algorithm": "Optimized with Cannon", "ExecutionTime": 0.000100},
```

```
    {"N": 10, "Processors": 16, "Algorithm": "Parallel", "ExecutionTime": 0.000058},
    {"N": 10, "Processors": 16, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.056614},
    {"N": 10, "Processors": 16, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.056822},
    {"N": 10, "Processors": 16, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.057961},
    {"N": 10, "Processors": 16, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.000003},
    {"N": 10, "Processors": 16, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.000113},


    {"N": 10, "Processors": 25, "Algorithm": "Parallel", "ExecutionTime": 0.000076},
    {"N": 10, "Processors": 25, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.062076},
    {"N": 10, "Processors": 25, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.061133},
    {"N": 10, "Processors": 25, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.063753},
    {"N": 10, "Processors": 25, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.051946},
    {"N": 10, "Processors": 25, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.374899},

  # N = 20 data

    {"N": 20, "Processors": 1, "Algorithm": "Parallel", "ExecutionTime": 0.000004},
    {"N": 20, "Processors": 1, "Algorithm": "Bit Array With Cyclic", "ExecutionTime": 0.020952},
    {"N": 20, "Processors": 1, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.021023},
    {"N": 20, "Processors": 1, "Algorithm": "Bit Array With Cannon", "ExecutionTime": 0.020032},
    {"N": 20, "Processors": 1, "Algorithm": "Optimized with Cyclic", "ExecutionTime": 0.000028},
    {"N": 20, "Processors": 1, "Algorithm": "Optimized with Cannon", "ExecutionTime": 0.000047},


    {"N": 20, "Processors": 4, "Algorithm": "Parallel", "ExecutionTime": 0.000009},
    {"N": 20, "Processors": 4, "Algorithm": "Bit Array With Cyclic", "ExecutionTime": 0.045682},
    {"N": 20, "Processors": 4, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.046516},
    {"N": 20, "Processors": 4, "Algorithm": "Bit Array With Cannon", "ExecutionTime": 0.047858},
    {"N": 20, "Processors": 4, "Algorithm": "Optimized with Cyclic", "ExecutionTime": 0.000009},
    {"N": 20, "Processors": 4, "Algorithm": "Optimized with Cannon", "ExecutionTime": 0.000077},
```

```
    {"N": 20, "Processors": 9, "Algorithm": "Parallel", "ExecutionTime": 0.000021},
    {"N": 20, "Processors": 9, "Algorithm": "Bit Array With Cyclic", "ExecutionTime": 0.054833},
    {"N": 20, "Processors": 9, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.053942},
    {"N": 20, "Processors": 9, "Algorithm": "Bit Array With Cannon", "ExecutionTime": 0.054350},
    {"N": 20, "Processors": 9, "Algorithm": "Optimized with Cyclic", "ExecutionTime": 0.000005},
    {"N": 20, "Processors": 9, "Algorithm": "Optimized with Cannon", "ExecutionTime": 0.000113},

    {"N": 20, "Processors": 16, "Algorithm": "Parallel", "ExecutionTime": 0.000018},
    {"N": 20, "Processors": 16, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.057629},
    {"N": 20, "Processors": 16, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.056972},
    {"N": 20, "Processors": 16, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.057371},
    {"N": 20, "Processors": 16, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.000004},
    {"N": 20, "Processors": 16, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.000116},

    {"N": 20, "Processors": 25, "Algorithm": "Parallel", "ExecutionTime": 0.000028},
    {"N": 20, "Processors": 25, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.062965},
    {"N": 20, "Processors": 25, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.061605},
    {"N": 20, "Processors": 25, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.063493},
    {"N": 20, "Processors": 25, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.051985},
    {"N": 20, "Processors": 25, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.288004},

  # N = 40 data
    {"N": 40, "Processors": 1, "Algorithm": "Parallel", "ExecutionTime": 0.000013},
    {"N": 40, "Processors": 1, "Algorithm": "Bit Array With Cyclic", "ExecutionTime": 0.020849},
    {"N": 40, "Processors": 1, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.022125},
    {"N": 40, "Processors": 1, "Algorithm": "Bit Array With Cannon", "ExecutionTime": 0.020051},
    {"N": 40, "Processors": 1, "Algorithm": "Optimized with Cyclic", "ExecutionTime": 0.000186},
    {"N": 40, "Processors": 1, "Algorithm": "Optimized with Cannon", "ExecutionTime": 0.000211},

    {"N": 40, "Processors": 4, "Algorithm": "Parallel", "ExecutionTime": 0.000018},
    {"N": 40, "Processors": 4, "Algorithm": "Bit Array With Cyclic", "ExecutionTime": 0.045937},
```

```
    {"N": 40, "Processors": 4, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.046399},
    {"N": 40, "Processors": 4, "Algorithm": "Bit Array With Cannon", "ExecutionTime": 0.048276},
    {"N": 40, "Processors": 4, "Algorithm": "Optimized with Cyclic", "ExecutionTime": 0.000045},
    {"N": 40, "Processors": 4, "Algorithm": "Optimized with Cannon", "ExecutionTime": 0.000379},

    {"N": 40, "Processors": 9, "Algorithm": "Parallel", "ExecutionTime": 0.000036},
    {"N": 40, "Processors": 9, "Algorithm": "Bit Array With Cyclic", "ExecutionTime": 0.054588},
    {"N": 40, "Processors": 9, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.053919},
    {"N": 40, "Processors": 9, "Algorithm": "Bit Array With Cannon", "ExecutionTime": 0.055243},
    {"N": 40, "Processors": 9, "Algorithm": "Optimized with Cyclic", "ExecutionTime": 0.000021},
    {"N": 40, "Processors": 9, "Algorithm": "Optimized with Cannon", "ExecutionTime": 0.000224},

    {"N": 40, "Processors": 16, "Algorithm": "Parallel", "ExecutionTime": 0.000051},
    {"N": 40, "Processors": 16, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.056907},
    {"N": 40, "Processors": 16, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.056717},
    {"N": 40, "Processors": 16, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.056940},
    {"N": 40, "Processors": 16, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.000013},
    {"N": 40, "Processors": 16, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.000188},

    {"N": 40, "Processors": 25, "Algorithm": "Parallel", "ExecutionTime": 0.000053},
    {"N": 40, "Processors": 25, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.061958},
    {"N": 40, "Processors": 25, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.062379},
    {"N": 40, "Processors": 25, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.066770},
    {"N": 40, "Processors": 25, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.051981},
    {"N": 40, "Processors": 25, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.273750},

  # N = 80 data
    {"N": 80, "Processors": 1, "Algorithm": "Parallel", "ExecutionTime": 0.000055},
    {"N": 80, "Processors": 1, "Algorithm": "Bit Array With Cyclic", "ExecutionTime": 0.020956},
    {"N": 80, "Processors": 1, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.021158},
    {"N": 80, "Processors": 1, "Algorithm": "Bit Array With Cannon", "ExecutionTime": 0.020547},
    {"N": 80, "Processors": 1, "Algorithm": "Optimized with Cyclic", "ExecutionTime": 0.001191},
```

```
    {"N": 80, "Processors": 1, "Algorithm": "Optimized with Cannon", "ExecutionTime": 0.001235},

    {"N": 80, "Processors": 4, "Algorithm": "Parallel", "ExecutionTime": 0.000090},
    {"N": 80, "Processors": 4, "Algorithm": "Bit Array With Cyclic", "ExecutionTime": 0.048596},
    {"N": 80, "Processors": 4, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.046387},
    {"N": 80, "Processors": 4, "Algorithm": "Bit Array With Cannon", "ExecutionTime": 0.047998},
    {"N": 80, "Processors": 4, "Algorithm": "Optimized with Cyclic", "ExecutionTime": 0.000319},
    {"N": 80, "Processors": 4, "Algorithm": "Optimized with Cannon", "ExecutionTime": 0.001422},

    {"N": 80, "Processors": 9, "Algorithm": "Parallel", "ExecutionTime": 0.000085},
    {"N": 80, "Processors": 9, "Algorithm": "Bit Array With Cyclic", "ExecutionTime": 0.054822},
    {"N": 80, "Processors": 9, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.053850},
    {"N": 80, "Processors": 9, "Algorithm": "Bit Array With Cannon", "ExecutionTime": 0.055000},
    {"N": 80, "Processors": 9, "Algorithm": "Optimized with Cyclic", "ExecutionTime": 0.000142},
    {"N": 80, "Processors": 9, "Algorithm": "Optimized with Cannon", "ExecutionTime": 0.000964},

    {"N": 80, "Processors": 16, "Algorithm": "Parallel", "ExecutionTime": 0.000127},
    {"N": 80, "Processors": 16, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.056294},
    {"N": 80, "Processors": 16, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.055819},
    {"N": 80, "Processors": 16, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.056944},
    {"N": 80, "Processors": 16, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.000077},
    {"N": 80, "Processors": 16, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.000707},

    {"N": 80, "Processors": 25, "Algorithm": "Parallel", "ExecutionTime": 0.000132},
    {"N": 80, "Processors": 25, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.061890},
    {"N": 80, "Processors": 25, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.061315},
    {"N": 80, "Processors": 25, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.066120},
    {"N": 80, "Processors": 25, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.051982},
    {"N": 80, "Processors": 25, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.202889},

  # N = 160 data
    {"N": 160, "Processors": 1, "Algorithm": "Parallel", "ExecutionTime": 0.000212},
```

```
      {"N": 160, "Processors": 1, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.020993},
      {"N": 160, "Processors": 1, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.021235},
      {"N": 160, "Processors": 1, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.020252},
      {"N": 160, "Processors": 1, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.007934},
      {"N": 160, "Processors": 1, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.008015},

      {"N": 160, "Processors": 4, "Algorithm": "Parallel", "ExecutionTime": 0.000251},
      {"N": 160, "Processors": 4, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.047229},
      {"N": 160, "Processors": 4, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.046423},
      {"N": 160, "Processors": 4, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.048368},
      {"N": 160, "Processors": 4, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.002079},
      {"N": 160, "Processors": 4, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.008036},

      {"N": 160, "Processors": 9, "Algorithm": "Parallel", "ExecutionTime": 0.000294},
      {"N": 160, "Processors": 9, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.054684},
      {"N": 160, "Processors": 9, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.054053},
      {"N": 160, "Processors": 9, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.054863},
      {"N": 160, "Processors": 9, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.000990},
      {"N": 160, "Processors": 9, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.005732},

      {"N": 160, "Processors": 16, "Algorithm": "Parallel", "ExecutionTime": 0.000280},
      {"N": 160, "Processors": 16, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.056106},
      {"N": 160, "Processors": 16, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.055964},
      {"N": 160, "Processors": 16, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.057182},
      {"N": 160, "Processors": 16, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
```

```
0.000575},
      {"N": 160, "Processors": 16, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.004040},

      {"N": 160, "Processors": 25, "Algorithm": "Parallel", "ExecutionTime": 0.000316},
      {"N": 160, "Processors": 25, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.061720},
      {"N": 160, "Processors": 25, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.064253},
      {"N": 160, "Processors": 25, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.067058},
      {"N": 160, "Processors": 25, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.051980},
      {"N": 160, "Processors": 25, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.207927},

    # N = 320 data
      {"N": 320, "Processors": 1, "Algorithm": "Parallel", "ExecutionTime": 0.000759},
      {"N": 320, "Processors": 1, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.021463},
      {"N": 320, "Processors": 1, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.021105},
      {"N": 320, "Processors": 1, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.020864},
      {"N": 320, "Processors": 1, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.056962},
      {"N": 320, "Processors": 1, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.122175},

      {"N": 320, "Processors": 4, "Algorithm": "Parallel", "ExecutionTime": 0.000887},
      {"N": 320, "Processors": 4, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.047700},
      {"N": 320, "Processors": 4, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.046908},
      {"N": 320, "Processors": 4, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.048792},
      {"N": 320, "Processors": 4, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.014328},
      {"N": 320, "Processors": 4, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.057067},

      {"N": 320, "Processors": 9, "Algorithm": "Parallel", "ExecutionTime": 0.000999},
```

```
      {"N": 320, "Processors": 9, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.054965},
      {"N": 320, "Processors": 9, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.054867},
      {"N": 320, "Processors": 9, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.055359},
      {"N": 320, "Processors": 9, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.006525},
      {"N": 320, "Processors": 9, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.038873},

      {"N": 320, "Processors": 16, "Algorithm": "Parallel", "ExecutionTime": 0.001036},
      {"N": 320, "Processors": 16, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.057251},
      {"N": 320, "Processors": 16, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.055913},
      {"N": 320, "Processors": 16, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.057039},
      {"N": 320, "Processors": 16, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.003740},
      {"N": 320, "Processors": 16, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.027479},

      {"N": 320, "Processors": 25, "Algorithm": "Parallel", "ExecutionTime": 0.001111},
      {"N": 320, "Processors": 25, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.062416},
      {"N": 320, "Processors": 25, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.064166},
      {"N": 320, "Processors": 25, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.067060},
      {"N": 320, "Processors": 25, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.051978},
      {"N": 320, "Processors": 25, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.207871},

    # N = 640 data
      {"N": 640, "Processors": 1, "Algorithm": "Parallel", "ExecutionTime": 0.002907},
      {"N": 640, "Processors": 1, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.022497},
      {"N": 640, "Processors": 1, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.022620},
      {"N": 640, "Processors": 1, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
```

```
0.022010},
      {"N": 640, "Processors": 1, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.421515},
      {"N": 640, "Processors": 1, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.849766},

      {"N": 640, "Processors": 4, "Algorithm": "Parallel", "ExecutionTime": 0.000887},
      {"N": 640, "Processors": 4, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.049157},
      {"N": 640, "Processors": 4, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.048694},
      {"N": 640, "Processors": 4, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.050724},
      {"N": 640, "Processors": 4, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.103540},
      {"N": 640, "Processors": 4, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.427194},

      {"N": 640, "Processors": 9, "Algorithm": "Parallel", "ExecutionTime": 0.003683},
      {"N": 640, "Processors": 9, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.056359},
      {"N": 640, "Processors": 9, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.055746},
      {"N": 640, "Processors": 9, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.057503},
      {"N": 640, "Processors": 9, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.046701},
      {"N": 640, "Processors": 9, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.287749},

      {"N": 640, "Processors": 16, "Algorithm": "Parallel", "ExecutionTime": 0.003717},
      {"N": 640, "Processors": 16, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.057632},
      {"N": 640, "Processors": 16, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.057385},
      {"N": 640, "Processors": 16, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.059067},
      {"N": 640, "Processors": 16, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.026160},
      {"N": 640, "Processors": 16, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.204131},
```

```
      {"N": 640, "Processors": 25, "Algorithm": "Parallel", "ExecutionTime": 0.003813},
      {"N": 640, "Processors": 25, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.063307},
      {"N": 640, "Processors": 25, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.065304},
      {"N": 640, "Processors": 25, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.068621},
      {"N": 640, "Processors": 25, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.079981},
      {"N": 640, "Processors": 25, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.383870},

    # n = 1000
      {"N": 1000, "Processors": 1, "Algorithm": "Parallel", "ExecutionTime": 0.007351},
      {"N": 1000, "Processors": 1, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.024891},
      {"N": 1000, "Processors": 1, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.024704},
      {"N": 1000, "Processors": 1, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.025393},
      {"N": 1000, "Processors": 1, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
1.545316},
      {"N": 1000, "Processors": 1, "Algorithm": "Optimized with Cannon", "ExecutionTime":
3.059727},

      {"N": 1000, "Processors": 4, "Algorithm": "Parallel", "ExecutionTime": 0.007249},
      {"N": 1000, "Processors": 4, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.051908},
      {"N": 1000, "Processors": 4, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.051710},
      {"N": 1000, "Processors": 4, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.054331},
      {"N": 1000, "Processors": 4, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.376487},
      {"N": 1000, "Processors": 4, "Algorithm": "Optimized with Cannon", "ExecutionTime":
1.583326},

      {"N": 1000, "Processors": 9, "Algorithm": "Parallel", "ExecutionTime": 0.008180},
      {"N": 1000, "Processors": 9, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.058844},
```

```
      {"N": 1000, "Processors": 9, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.058544},
      {"N": 1000, "Processors": 9, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.060911},
      {"N": 1000, "Processors": 9, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.168908},
      {"N": 1000, "Processors": 9, "Algorithm": "Optimized with Cannon", "ExecutionTime":
1.063261},

      {"N": 1000, "Processors": 16, "Algorithm": "Parallel", "ExecutionTime": 0.008197},
      {"N": 1000, "Processors": 16, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.060397},
      {"N": 1000, "Processors": 16, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.059617},
      {"N": 1000, "Processors": 16, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.061755},
      {"N": 1000, "Processors": 16, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.094568},
      {"N": 1000, "Processors": 16, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.757075},

      {"N": 1000, "Processors": 25, "Algorithm": "Parallel", "ExecutionTime": 0.008477},
      {"N": 1000, "Processors": 25, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.066083},
      {"N": 1000, "Processors": 25, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.067825},
      {"N": 1000, "Processors": 25, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.072130},
      {"N": 1000, "Processors": 25, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.147589},
      {"N": 1000, "Processors": 25, "Algorithm": "Optimized with Cannon", "ExecutionTime":
0.738922},

    # N = 2000 data
      {"N": 2000, "Processors": 1, "Algorithm": "Parallel", "ExecutionTime": 0.027748},
      {"N": 2000, "Processors": 1, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.036384},
      {"N": 2000, "Processors": 1, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.036403},
      {"N": 2000, "Processors": 1, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
```

```
0.036016},
      {"N": 2000, "Processors": 1, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
11.830161},
      {"N": 2000, "Processors": 1, "Algorithm": "Optimized with Cannon", "ExecutionTime":
23.451408},

      {"N": 2000, "Processors": 4, "Algorithm": "Parallel", "ExecutionTime": 0.028115},
      {"N": 2000, "Processors": 4, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.065504},
      {"N": 2000, "Processors": 4, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.066725},
      {"N": 2000, "Processors": 4, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.072387},
      {"N": 2000, "Processors": 4, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
2.907338},
      {"N": 2000, "Processors": 4, "Algorithm": "Optimized with Cannon", "ExecutionTime":
12.264726},

      {"N": 2000, "Processors": 9, "Algorithm": "Parallel", "ExecutionTime": 0.031134},
      {"N": 2000, "Processors": 9, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.072462},
      {"N": 2000, "Processors": 9, "Algorithm": "Bit Array With Block", "ExecutionTime": 0.07133},
      {"N": 2000, "Processors": 9, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.078214},
      {"N": 2000, "Processors": 9, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
1.282809},
      {"N": 2000, "Processors": 9, "Algorithm": "Optimized with Cannon", "ExecutionTime":
8.184642},

      {"N": 2000, "Processors": 16, "Algorithm": "Parallel", "ExecutionTime": 0.030612},
      {"N": 2000, "Processors": 16, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.072946},
      {"N": 2000, "Processors": 16, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.072597},
      {"N": 2000, "Processors": 16, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.078276},
      {"N": 2000, "Processors": 16, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.726995},
      {"N": 2000, "Processors": 16, "Algorithm": "Optimized with Cannon", "ExecutionTime":
5.866201},
```

```
    {"N": 2000, "Processors": 25, "Algorithm": "Parallel", "ExecutionTime": 0.031861},
    {"N": 2000, "Processors": 25, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.078022},
    {"N": 2000, "Processors": 25, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.078894},
    {"N": 2000, "Processors": 25, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.085580},
    {"N": 2000, "Processors": 25, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
0.974470},
    {"N": 2000, "Processors": 25, "Algorithm": "Optimized with Cannon", "ExecutionTime":
4.852891},

  # N = 8000 data
    {"N": 8000, "Processors": 1, "Algorithm": "Parallel", "ExecutionTime": 0.504271},
    {"N": 8000, "Processors": 1, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.275613},
    {"N": 8000, "Processors": 1, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.303291},
    {"N": 8000, "Processors": 1, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.283631},
    {"N": 8000, "Processors": 1, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
1348.423483},
    {"N": 8000, "Processors": 1, "Algorithm": "Optimized with Cannon", "ExecutionTime":
1417.156170},

    {"N": 8000, "Processors": 4, "Algorithm": "Parallel", "ExecutionTime": 0.487673},
    {"N": 8000, "Processors": 4, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.347372},
    {"N": 8000, "Processors": 4, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.376975},
    {"N": 8000, "Processors": 4, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.493007},
    {"N": 8000, "Processors": 4, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
178.898074},
    {"N": 8000, "Processors": 4, "Algorithm": "Optimized with Cannon", "ExecutionTime":
758.687427},

    {"N": 8000, "Processors": 9, "Algorithm": "Parallel", "ExecutionTime": 0.478143},
    {"N": 8000, "Processors": 9, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
```

```
0.324421},
      {"N": 8000, "Processors": 9, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.336872},
      {"N": 8000, "Processors": 9, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.442236},
      {"N": 8000, "Processors": 9, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
78.607553},
      {"N": 8000, "Processors": 9, "Algorithm": "Optimized with Cannon", "ExecutionTime":
503.968842},

      {"N": 8000, "Processors": 16, "Algorithm": "Parallel", "ExecutionTime": 0.472326},
      {"N": 8000, "Processors": 16, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.316690},
      {"N": 8000, "Processors": 16, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.321481},
      {"N": 8000, "Processors": 16, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.407310},
      {"N": 8000, "Processors": 16, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
44.740368},
      {"N": 8000, "Processors": 16, "Algorithm": "Optimized with Cannon", "ExecutionTime":
362.913111},

      {"N": 8000, "Processors": 25, "Algorithm": "Parallel", "ExecutionTime": 0.492784},
      {"N": 8000, "Processors": 25, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
0.316568},
      {"N": 8000, "Processors": 25, "Algorithm": "Bit Array With Block", "ExecutionTime":
0.318291},
      {"N": 8000, "Processors": 25, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
0.385985},
      {"N": 8000, "Processors": 25, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
39.882981},
      {"N": 8000, "Processors": 25, "Algorithm": "Optimized with Cannon", "ExecutionTime":
296.455811},

    # N = 16000 data
      {"N": 16000, "Processors": 1, "Algorithm": "Parallel", "ExecutionTime": 2.704892},
      {"N": 16000, "Processors": 1, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
1.095530},
      {"N": 16000, "Processors": 1, "Algorithm": "Bit Array With Block", "ExecutionTime":
1.118999},
```

```
      {"N": 16000, "Processors": 1, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
1.311387},
      {"N": 16000, "Processors": 1, "Algorithm": "Optimized with Cyclic", "ExecutionTime": "DNF"},
      {"N": 16000, "Processors": 1, "Algorithm": "Optimized with Cannon", "ExecutionTime": "DNF"},

      {"N": 16000, "Processors": 4, "Algorithm": "Parallel", "ExecutionTime": 2.111424},
      {"N": 16000, "Processors": 4, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
1.368232},
      {"N": 16000, "Processors": 4, "Algorithm": "Bit Array With Block", "ExecutionTime":
1.534098},
      {"N": 16000, "Processors": 4, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
2.293561},
      {"N": 16000, "Processors": 4, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
1416.099513},
      {"N": 16000, "Processors": 4, "Algorithm": "Optimized with Cannon", "ExecutionTime": "DNF"},

      {"N": 16000, "Processors": 9, "Algorithm": "Parallel", "ExecutionTime": 1.969355},
      {"N": 16000, "Processors": 9, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
1.267211},
      {"N": 16000, "Processors": 9, "Algorithm": "Bit Array With Block", "ExecutionTime":
1.315586},
      {"N": 16000, "Processors": 9, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
2.087108},
      {"N": 16000, "Processors": 9, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
621.847429},
      {"N": 16000, "Processors": 9, "Algorithm": "Optimized with Cannon", "ExecutionTime": "DNF"},

      {"N": 16000, "Processors": 16, "Algorithm": "Parallel", "ExecutionTime": 1.889844},
      {"N": 16000, "Processors": 16, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
1.239115},
      {"N": 16000, "Processors": 16, "Algorithm": "Bit Array With Block", "ExecutionTime":
1.254280},
      {"N": 16000, "Processors": 16, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
2.153650},
      {"N": 16000, "Processors": 16, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
354.241536},
      {"N": 16000, "Processors": 16, "Algorithm": "Optimized with Cannon", "ExecutionTime":
2882.078682},

      {"N": 16000, "Processors": 25, "Algorithm": "Parallel", "ExecutionTime": 1.974310},
```

```
      {"N": 16000, "Processors": 25, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
1.230493},
      {"N": 16000, "Processors": 25, "Algorithm": "Bit Array With Block", "ExecutionTime":
1.239478},
      {"N": 16000, "Processors": 25, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
2.019938},
      {"N": 16000, "Processors": 25, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
303.418342},
      {"N": 16000, "Processors": 25, "Algorithm": "Optimized with Cannon", "ExecutionTime":
3147.422869},

    # N = 32000 data
      {"N": 32000, "Processors": 1, "Algorithm": "Parallel", "ExecutionTime": 13.987126},
      {"N": 32000, "Processors": 1, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
4.512644},
      {"N": 32000, "Processors": 1, "Algorithm": "Bit Array With Block", "ExecutionTime":
7.001417},
      {"N": 32000, "Processors": 1, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
5.560011},
      {"N": 32000, "Processors": 1, "Algorithm": "Optimized with Cyclic", "ExecutionTime": "DNF"},
      {"N": 32000, "Processors": 1, "Algorithm": "Optimized with Cannon", "ExecutionTime": "DNF"},

      {"N": 32000, "Processors": 4, "Algorithm": "Parallel", "ExecutionTime": 8.687078},
      {"N": 32000, "Processors": 4, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
5.440158},
      {"N": 32000, "Processors": 4, "Algorithm": "Bit Array With Block", "ExecutionTime":
6.141975},
      {"N": 32000, "Processors": 4, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
8.631922},
      {"N": 32000, "Processors": 4, "Algorithm": "Optimized with Cyclic", "ExecutionTime": "DNF"},
      {"N": 32000, "Processors": 4, "Algorithm": "Optimized with Cannon", "ExecutionTime": "DNF"},

      {"N": 32000, "Processors": 9, "Algorithm": "Parallel", "ExecutionTime": 7.911520},
      {"N": 32000, "Processors": 9, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
5.070082},
      {"N": 32000, "Processors": 9, "Algorithm": "Bit Array With Block", "ExecutionTime":
5.173290},
      {"N": 32000, "Processors": 9, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
9.983202},
      {"N": 32000, "Processors": 9, "Algorithm": "Optimized with Cyclic", "ExecutionTime": "DNF"},
```

```
      {"N": 32000, "Processors": 9, "Algorithm": "Optimized with Cannon", "ExecutionTime": "DNF"},

      {"N": 32000, "Processors": 16, "Algorithm": "Parallel", "ExecutionTime": 8.173827},
      {"N": 32000, "Processors": 16, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
4.828183},
      {"N": 32000, "Processors": 16, "Algorithm": "Bit Array With Block", "ExecutionTime":
4.877488},
      {"N": 32000, "Processors": 16, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
11.366624},
      {"N": 32000, "Processors": 16, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
2812.784026},
      {"N": 32000, "Processors": 16, "Algorithm": "Optimized with Cannon", "ExecutionTime":
"DNF"},

      {"N": 32000, "Processors": 25, "Algorithm": "Parallel", "ExecutionTime": 7.871757},
      {"N": 32000, "Processors": 25, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
4.786187},
      {"N": 32000, "Processors": 25, "Algorithm": "Bit Array With Block", "ExecutionTime":
4.909442},
      {"N": 32000, "Processors": 25, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
12.230786},
      {"N": 32000, "Processors": 25, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
2951.085973},
      {"N": 32000, "Processors": 25, "Algorithm": "Optimized with Cannon", "ExecutionTime":
"DNF"},

    # N = 40000 data
      {"N": 40000, "Processors": 1, "Algorithm": "Parallel", "ExecutionTime": 22.430775},
      {"N": 40000, "Processors": 1, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
6.968171},
      {"N": 40000, "Processors": 1, "Algorithm": "Bit Array With Block", "ExecutionTime":
9.578330},
      {"N": 40000, "Processors": 1, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
8.469553},
      {"N": 40000, "Processors": 1, "Algorithm": "Optimized with Cyclic", "ExecutionTime": "DNF"},
      {"N": 40000, "Processors": 1, "Algorithm": "Optimized with Cannon", "ExecutionTime": "DNF"},

      {"N": 40000, "Processors": 4, "Algorithm": "Parallel", "ExecutionTime": 13.654003},
      {"N": 40000, "Processors": 4, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
8.489121},
```

```
      {"N": 40000, "Processors": 4, "Algorithm": "Bit Array With Block", "ExecutionTime":
9.608803},
      {"N": 40000, "Processors": 4, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
13.606368},
      {"N": 40000, "Processors": 4, "Algorithm": "Optimized with Cyclic", "ExecutionTime": "DNF"},
      {"N": 40000, "Processors": 4, "Algorithm": "Optimized with Cannon", "ExecutionTime": "DNF"},

      {"N": 40000, "Processors": 9, "Algorithm": "Parallel", "ExecutionTime": 12.529540},
      {"N": 40000, "Processors": 9, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
7.798529},
      {"N": 40000, "Processors": 9, "Algorithm": "Bit Array With Block", "ExecutionTime":
8.065470},
      {"N": 40000, "Processors": 9, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
15.882469},
      {"N": 40000, "Processors": 9, "Algorithm": "Optimized with Cyclic", "ExecutionTime": "DNF"},
      {"N": 40000, "Processors": 9, "Algorithm": "Optimized with Cannon", "ExecutionTime": "DNF"},

      {"N": 40000, "Processors": 16, "Algorithm": "Parallel", "ExecutionTime": 13.253679},
      {"N": 40000, "Processors": 16, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
7.437856},
      {"N": 40000, "Processors": 16, "Algorithm": "Bit Array With Block", "ExecutionTime":
7.511777},
      {"N": 40000, "Processors": 16, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
18.221498},
      {"N": 40000, "Processors": 16, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
"DNF"},
      {"N": 40000, "Processors": 16, "Algorithm": "Optimized with Cannon", "ExecutionTime":
"DNF"},

      {"N": 40000, "Processors": 25, "Algorithm": "Parallel", "ExecutionTime": 12.633755},
      {"N": 40000, "Processors": 25, "Algorithm": "Bit Array With Cyclic", "ExecutionTime":
7.396951},
      {"N": 40000, "Processors": 25, "Algorithm": "Bit Array With Block", "ExecutionTime":
7.465304},
      {"N": 40000, "Processors": 25, "Algorithm": "Bit Array With Cannon", "ExecutionTime":
20.457722},
      {"N": 40000, "Processors": 25, "Algorithm": "Optimized with Cyclic", "ExecutionTime":
"DNF"},
      {"N": 40000, "Processors": 25, "Algorithm": "Optimized with Cannon", "ExecutionTime":
"DNF"},
```

```
    # N = 50,000 data
        {"Algorithm": "Parallel", "N": 50000, "Processors": 1, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 50000, "Processors": 1, "ExecutionTime":
11.076311},
        {"Algorithm": "Bit Array With Block", "N": 50000, "Processors": 1, "ExecutionTime":
17.318775},
        {"Algorithm": "Bit Array With Cannon", "N": 50000, "Processors": 1, "ExecutionTime":
13.555085},
        {"Algorithm": "Optimized with Cyclic", "N": 50000, "Processors": 1, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 50000, "Processors": 1, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 50000, "Processors": 4, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 50000, "Processors": 4, "ExecutionTime":
13.234685},
        {"Algorithm": "Bit Array With Block", "N": 50000, "Processors": 4, "ExecutionTime":
15.081690},
        {"Algorithm": "Bit Array With Cannon", "N": 50000, "Processors": 4, "ExecutionTime":
21.451990},
        {"Algorithm": "Optimized with Cyclic", "N": 50000, "Processors": 4, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 50000, "Processors": 4, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 50000, "Processors": 9, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 50000, "Processors": 9, "ExecutionTime":
11.999548},
        {"Algorithm": "Bit Array With Block", "N": 50000, "Processors": 9, "ExecutionTime":
12.576673},
        {"Algorithm": "Bit Array With Cannon", "N": 50000, "Processors": 9, "ExecutionTime":
26.091551},
        {"Algorithm": "Optimized with Cyclic", "N": 50000, "Processors": 9, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 50000, "Processors": 9, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 50000, "Processors": 16, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 50000, "Processors": 16, "ExecutionTime":
```

```
11.499121},
        {"Algorithm": "Bit Array With Block", "N": 50000, "Processors": 16, "ExecutionTime":
11.707929},
        {"Algorithm": "Bit Array With Cannon", "N": 50000, "Processors": 16, "ExecutionTime":
29.428584},
        {"Algorithm": "Optimized with Cyclic", "N": 50000, "Processors": 16, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 50000, "Processors": 16, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 50000, "Processors": 25, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 50000, "Processors": 25, "ExecutionTime":
11.433776},
        {"Algorithm": "Bit Array With Block", "N": 50000, "Processors": 25, "ExecutionTime":
11.430980},
        {"Algorithm": "Bit Array With Cannon", "N": 50000, "Processors": 25, "ExecutionTime":
33.139867},
        {"Algorithm": "Optimized with Cyclic", "N": 50000, "Processors": 25, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 50000, "Processors": 25, "ExecutionTime":
"DNF"},

    # N = 60,000 data
        {"Algorithm": "Parallel", "N": 60000, "Processors": 1, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 60000, "Processors": 1, "ExecutionTime":
15.961117},
        {"Algorithm": "Bit Array With Block", "N": 60000, "Processors": 1, "ExecutionTime":
24.338447},
        {"Algorithm": "Bit Array With Cannon", "N": 60000, "Processors": 1, "ExecutionTime":
22.102536},
        {"Algorithm": "Optimized with Cyclic", "N": 60000, "Processors": 1, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 60000, "Processors": 1, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 60000, "Processors": 4, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 60000, "Processors": 4, "ExecutionTime":
19.008338},
        {"Algorithm": "Bit Array With Block", "N": 60000, "Processors": 4, "ExecutionTime":
21.743348},
```

```
        {"Algorithm": "Bit Array With Cannon", "N": 60000, "Processors": 4, "ExecutionTime":
30.991873},
        {"Algorithm": "Optimized with Cyclic", "N": 60000, "Processors": 4, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 60000, "Processors": 4, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 60000, "Processors": 9, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 60000, "Processors": 9, "ExecutionTime":
17.232217},
        {"Algorithm": "Bit Array With Block", "N": 60000, "Processors": 9, "ExecutionTime":
18.088442},
        {"Algorithm": "Bit Array With Cannon", "N": 60000, "Processors": 9, "ExecutionTime":
37.668923},
        {"Algorithm": "Optimized with Cyclic", "N": 60000, "Processors": 9, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 60000, "Processors": 9, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 60000, "Processors": 16, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 60000, "Processors": 16, "ExecutionTime":
16.430711},
        {"Algorithm": "Bit Array With Block", "N": 60000, "Processors": 16, "ExecutionTime":
16.833956},
        {"Algorithm": "Bit Array With Cannon", "N": 60000, "Processors": 16, "ExecutionTime":
43.542032},
        {"Algorithm": "Optimized with Cyclic", "N": 60000, "Processors": 16, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 60000, "Processors": 16, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 60000, "Processors": 25, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 60000, "Processors": 25, "ExecutionTime":
16.339115},
        {"Algorithm": "Bit Array With Block", "N": 60000, "Processors": 25, "ExecutionTime":
16.622011},
        {"Algorithm": "Bit Array With Cannon", "N": 60000, "Processors": 25, "ExecutionTime":
48.515912},
        {"Algorithm": "Optimized with Cyclic", "N": 60000, "Processors": 25, "ExecutionTime":
"DNF"},
```

```
        {"Algorithm": "Optimized with Cannon", "N": 60000, "Processors": 25, "ExecutionTime":
"DNF"},

    # N = 70,000 data
        {"Algorithm": "Parallel", "N": 70000, "Processors": 1, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 70000, "Processors": 1, "ExecutionTime":
21.604208},
        {"Algorithm": "Bit Array With Block", "N": 70000, "Processors": 1, "ExecutionTime":
34.300779},
        {"Algorithm": "Bit Array With Cannon", "N": 70000, "Processors": 1, "ExecutionTime":
30.312118},
        {"Algorithm": "Optimized with Cyclic", "N": 70000, "Processors": 1, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 70000, "Processors": 1, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 70000, "Processors": 4, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 70000, "Processors": 4, "ExecutionTime":
25.837963},
        {"Algorithm": "Bit Array With Block", "N": 70000, "Processors": 4, "ExecutionTime":
29.653188},
        {"Algorithm": "Bit Array With Cannon", "N": 70000, "Processors": 4, "ExecutionTime":
42.428553},
        {"Algorithm": "Optimized with Cyclic", "N": 70000, "Processors": 4, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 70000, "Processors": 4, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 70000, "Processors": 9, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 70000, "Processors": 9, "ExecutionTime":
23.217536},
        {"Algorithm": "Bit Array With Block", "N": 70000, "Processors": 9, "ExecutionTime":
24.608068},
        {"Algorithm": "Bit Array With Cannon", "N": 70000, "Processors": 9, "ExecutionTime":
52.038788},
        {"Algorithm": "Optimized with Cyclic", "N": 70000, "Processors": 9, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 70000, "Processors": 9, "ExecutionTime":
"DNF"},
```

```
        {"Algorithm": "Parallel", "N": 70000, "Processors": 16, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 70000, "Processors": 16, "ExecutionTime":
22.237129},
        {"Algorithm": "Bit Array With Block", "N": 70000, "Processors": 16, "ExecutionTime":
22.876571},
        {"Algorithm": "Bit Array With Cannon", "N": 70000, "Processors": 16, "ExecutionTime":
60.178705},
        {"Algorithm": "Optimized with Cyclic", "N": 70000, "Processors": 16, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 70000, "Processors": 16, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 70000, "Processors": 25, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 70000, "Processors": 25, "ExecutionTime":
22.120452},
        {"Algorithm": "Bit Array With Block", "N": 70000, "Processors": 25, "ExecutionTime":
22.583568},
        {"Algorithm": "Bit Array With Cannon", "N": 70000, "Processors": 25, "ExecutionTime":
68.533848},
        {"Algorithm": "Optimized with Cyclic", "N": 70000, "Processors": 25, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 70000, "Processors": 25, "ExecutionTime":
"DNF"},

    # N = 80,000 data
        {"Algorithm": "Parallel", "N": 80000, "Processors": 1, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 80000, "Processors": 1, "ExecutionTime":
28.152983},
        {"Algorithm": "Bit Array With Block", "N": 80000, "Processors": 1, "ExecutionTime":
38.913345},
        {"Algorithm": "Bit Array With Cannon", "N": 80000, "Processors": 1, "ExecutionTime":
40.043552},
        {"Algorithm": "Optimized with Cyclic", "N": 80000, "Processors": 1, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 80000, "Processors": 1, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 80000, "Processors": 4, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 80000, "Processors": 4, "ExecutionTime":
33.793175},
```

```
        {"Algorithm": "Bit Array With Block", "N": 80000, "Processors": 4, "ExecutionTime":
38.787977},
        {"Algorithm": "Bit Array With Cannon", "N": 80000, "Processors": 4, "ExecutionTime":
55.367532},
        {"Algorithm": "Optimized with Cyclic", "N": 80000, "Processors": 4, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 80000, "Processors": 4, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 80000, "Processors": 9, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 80000, "Processors": 9, "ExecutionTime":
30.370068},
        {"Algorithm": "Bit Array With Block", "N": 80000, "Processors": 9, "ExecutionTime":
32.136284},
        {"Algorithm": "Bit Array With Cannon", "N": 80000, "Processors": 9, "ExecutionTime":
69.795520},
        {"Algorithm": "Optimized with Cyclic", "N": 80000, "Processors": 9, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 80000, "Processors": 9, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 80000, "Processors": 16, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 80000, "Processors": 16, "ExecutionTime":
28.938943},
        {"Algorithm": "Bit Array With Block", "N": 80000, "Processors": 16, "ExecutionTime":
29.859504},
        {"Algorithm": "Bit Array With Cannon", "N": 80000, "Processors": 16, "ExecutionTime":
79.91889},
        {"Algorithm": "Optimized with Cyclic", "N": 80000, "Processors": 16, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 80000, "Processors": 16, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 80000, "Processors": 25, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 80000, "Processors": 25, "ExecutionTime":
28.894029},
        {"Algorithm": "Bit Array With Block", "N": 80000, "Processors": 25, "ExecutionTime":
29.545236},
        {"Algorithm": "Bit Array With Cannon", "N": 80000, "Processors": 25, "ExecutionTime":
90.702026},
```

```
        {"Algorithm": "Optimized with Cyclic", "N": 80000, "Processors": 25, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 80000, "Processors": 25, "ExecutionTime":
"DNF"},

        # N = 90,000 data
        {"Algorithm": "Parallel", "N": 90000, "Processors": 1, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 90000, "Processors": 1, "ExecutionTime":
35.438466},
        {"Algorithm": "Bit Array With Block", "N": 90000, "Processors": 1, "ExecutionTime":
54.701366},
        {"Algorithm": "Bit Array With Cannon", "N": 90000, "Processors": 1, "ExecutionTime":
46.073131},
        {"Algorithm": "Optimized with Cyclic", "N": 90000, "Processors": 1, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 90000, "Processors": 1, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 90000, "Processors": 4, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 90000, "Processors": 4, "ExecutionTime":
42.648385},
        {"Algorithm": "Bit Array With Block", "N": 90000, "Processors": 4, "ExecutionTime":
49.130687},
        {"Algorithm": "Bit Array With Cannon", "N": 90000, "Processors": 4, "ExecutionTime":
70.188177},
        {"Algorithm": "Optimized with Cyclic", "N": 90000, "Processors": 4, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 90000, "Processors": 4, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 90000, "Processors": 9, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 90000, "Processors": 9, "ExecutionTime":
38.140889},
        {"Algorithm": "Bit Array With Block", "N": 90000, "Processors": 9, "ExecutionTime":
40.632021},
        {"Algorithm": "Bit Array With Cannon", "N": 90000, "Processors": 9, "ExecutionTime":
86.298726},
        {"Algorithm": "Optimized with Cyclic", "N": 90000, "Processors": 9, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 90000, "Processors": 9, "ExecutionTime":
```

```
"DNF"},

        {"Algorithm": "Parallel", "N": 90000, "Processors": 16, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 90000, "Processors": 16, "ExecutionTime":
36.515410},
        {"Algorithm": "Bit Array With Block", "N": 90000, "Processors": 16, "ExecutionTime":
37.747332},
        {"Algorithm": "Bit Array With Cannon", "N": 90000, "Processors": 16, "ExecutionTime":
102.003456},
        {"Algorithm": "Optimized with Cyclic", "N": 90000, "Processors": 16, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 90000, "Processors": 16, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 90000, "Processors": 25, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 90000, "Processors": 25, "ExecutionTime":
36.526583},
        {"Algorithm": "Bit Array With Block", "N": 90000, "Processors": 25, "ExecutionTime":
37.386652},
        {"Algorithm": "Bit Array With Cannon", "N": 90000, "Processors": 25, "ExecutionTime":
115.222522},
        {"Algorithm": "Optimized with Cyclic", "N": 90000, "Processors": 25, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 90000, "Processors": 25, "ExecutionTime":
"DNF"},

    # N = 100,000 data
        {"Algorithm": "Parallel", "N": 100000, "Processors": 1, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 100000, "Processors": 1, "ExecutionTime":
44.150632},
        {"Algorithm": "Bit Array With Block", "N": 100000, "Processors": 1, "ExecutionTime":
62.416038},
        {"Algorithm": "Bit Array With Cannon", "N": 100000, "Processors": 1, "ExecutionTime":
53.234932},
        {"Algorithm": "Optimized with Cyclic", "N": 100000, "Processors": 1, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 100000, "Processors": 1, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 100000, "Processors": 4, "ExecutionTime": "DNF"},
```

```
        {"Algorithm": "Bit Array With Cyclic", "N": 100000, "Processors": 4, "ExecutionTime":
52.555800},
        {"Algorithm": "Bit Array With Block", "N": 100000, "Processors": 4, "ExecutionTime":
60.704886},
        {"Algorithm": "Bit Array With Cannon", "N": 100000, "Processors": 4, "ExecutionTime":
86.729071},
        {"Algorithm": "Optimized with Cyclic", "N": 100000, "Processors": 4, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 100000, "Processors": 4, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 100000, "Processors": 9, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 100000, "Processors": 9, "ExecutionTime":
47.258143},
        {"Algorithm": "Bit Array With Block", "N": 100000, "Processors": 9, "ExecutionTime":
50.165235},
        {"Algorithm": "Bit Array With Cannon", "N": 100000, "Processors": 9, "ExecutionTime":
107.741419},
        {"Algorithm": "Optimized with Cyclic", "N": 100000, "Processors": 9, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 100000, "Processors": 9, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 100000, "Processors": 16, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 100000, "Processors": 16, "ExecutionTime":
44.959808},
        {"Algorithm": "Bit Array With Block", "N": 100000, "Processors": 16, "ExecutionTime":
46.585156},
        {"Algorithm": "Bit Array With Cannon", "N": 100000, "Processors": 16, "ExecutionTime":
127.212180},
        {"Algorithm": "Optimized with Cyclic", "N": 100000, "Processors": 16, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 100000, "Processors": 16, "ExecutionTime":
"DNF"},

        {"Algorithm": "Parallel", "N": 100000, "Processors": 25, "ExecutionTime": "DNF"},
        {"Algorithm": "Bit Array With Cyclic", "N": 100000, "Processors": 25, "ExecutionTime":
44.902574},
        {"Algorithm": "Bit Array With Block", "N": 100000, "Processors": 25, "ExecutionTime":
46.189385},
```

```
        {"Algorithm": "Bit Array With Cannon", "N": 100000, "Processors": 25, "ExecutionTime":
147.773271},
        {"Algorithm": "Optimized with Cyclic", "N": 100000, "Processors": 25, "ExecutionTime":
"DNF"},
        {"Algorithm": "Optimized with Cannon", "N": 100000, "Processors": 25, "ExecutionTime":
"DNF"}
    ])

    # Convert to DataFrame
    df = pd.DataFrame(data)

    # Handle 'DNF' values
    df['ExecutionTime'] = df['ExecutionTime'].replace('DNF', np.nan)

    return df

# Load data
df = prepare_benchmark_data()

# View the data structure
print("Data structure:")
print(df.head())

# Analysis 1: Compare execution time by problem size for each algorithm (fixed processor count)
def plot_execution_vs_problem_size(processor_count=1):
    """Plot execution time vs problem size for each algorithm with fixed processor count."""
    plt.figure(figsize=(12, 8))

    # Filter data for the specified processor count
    data = df[df['Processors'] == processor_count]

    # Check if data exists
    if data.empty:
        print(f"No data available for processor count {processor_count}")
        return plt

    # Create a pivot table: columns are algorithms, rows are N values
    pivot_data = data.pivot(index='N', columns='Algorithm', values='ExecutionTime')

    # Plot
```

```
    for col in pivot_data.columns:
        # Skip columns with all NaN values and drop NaN values when plotting
        if pivot_data[col].notna().any():
            valid_data = pivot_data[col].dropna()
            plt.plot(valid_data.index, valid_data.values, marker='o', linewidth=2, label=col)

    plt.title(f'Execution Time vs Problem Size (Processors={processor_count})')
    plt.xlabel('Problem Size (N)')
    plt.ylabel('Execution Time (seconds)')
    plt.xscale('log')
    plt.yscale('log')
    plt.grid(True, which="both", ls="-")
    plt.legend()
    plt.tight_layout()

    return plt

# Analysis 2: Compare execution time by processor count for each algorithm (fixed problem size)
def plot_execution_vs_processors(problem_size=1000):
    """Plot execution time vs processor count for each algorithm with fixed problem size."""
    plt.figure(figsize=(12, 8))

    # Filter data for the specified problem size
    data = df[df['N'] == problem_size]

    # Create a pivot table: columns are algorithms, rows are processor counts
    pivot_data = data.pivot(index='Processors', columns='Algorithm', values='ExecutionTime')

    # Plot
    for col in pivot_data.columns:
        plt.plot(pivot_data.index, pivot_data[col], marker='o', linewidth=2, label=col)

    plt.title(f'Execution Time vs Number of Processors (N={problem_size})')
    plt.xlabel('Number of Processors')
    plt.ylabel('Execution Time (seconds)')
    plt.grid(True)
    plt.legend()
    plt.tight_layout()

    return plt
```

```
# Analysis 3: Calculate speedup and parallel efficiency
def calculate_parallel_metrics():
    """Calculate speedup and parallel efficiency for different algorithms and problem sizes."""
    # Group data by Algorithm and N
    metrics = []

    for algo in df['Algorithm'].unique():
        for n in df['N'].unique():
            # Get the execution time with 1 processor (serial time)
            serial_data = df[(df['Algorithm'] == algo) & (df['N'] == n) & (df['Processors'] == 1)]

            if len(serial_data) == 0:
                continue

            serial_time = serial_data['ExecutionTime'].values[0]

            # Calculate speedup and efficiency for each processor count
            for proc in [4, 9, 16, 25]:
                parallel_data = df[(df['Algorithm'] == algo) & (df['N'] == n) & (df['Processors']
== proc)]

                if len(parallel_data) == 0:
                    continue

                parallel_time = parallel_data['ExecutionTime'].values[0]

                # Calculate metrics
                speedup = serial_time / parallel_time
                efficiency = speedup / proc * 100  # as percentage

                metrics.append({
                    'Algorithm': algo,
                    'N': n,
                    'Processors': proc,
                    'Speedup': speedup,
                    'Efficiency': efficiency
                })

    return pd.DataFrame(metrics)
```

```
# Analysis 4: Heatmap of execution times across problem sizes and processor counts
def plot_heatmap(algorithm):
    """Create a heatmap of execution times for a specific algorithm."""
    # Filter data for the specified algorithm
    data = df[df['Algorithm'] == algorithm]

    # Create a pivot table
    pivot_data = data.pivot(index='Processors', columns='N', values='ExecutionTime')

    # Plot heatmap
    plt.figure(figsize=(16, 10))  # Increased figure size

    # Custom formatter for annotations based on value range
    def fmt_func(val):
        if val < 0.001:
            return f'{val:.2e}'  # Scientific notation for very small values
        elif val < 0.1:
            return f'{val:.3f}'  # 3 decimal places for small values
        elif val < 1:
            return f'{val:.2f}'  # 2 decimal places for medium values
        else:
            return f'{val:.1f}'  # 1 decimal place for large values

    # Create heatmap with customized annotations
    ax = sns.heatmap(
        pivot_data,
        annot=True,
        fmt='',
        cmap='viridis',
        norm=LogNorm(),
        cbar_kws={'label': 'Execution Time (seconds)'},
        annot_kws={'fontsize': 8}  # Smaller font size for annotations
    )

    # Apply the custom formatter to annotations
    for t in ax.texts:
        val = float(t.get_text())
        t.set_text(fmt_func(val))
```

```
    # Improve axis labels and title
    plt.title(f'Execution Time Heatmap for {algorithm}', fontsize=14)
    plt.xlabel('Problem Size (N)', fontsize=12)
    plt.ylabel('Number of Processors', fontsize=12)

    # Rotate x-axis labels for better readability
    plt.xticks(rotation=45, ha='right')

    plt.tight_layout()
    return plt

# Run analyses and display plots
print("\nExecution Time vs Problem Size (1 processor):")
plot1 = plot_execution_vs_problem_size(1)
plt.show()

print("\nExecution Time vs Problem Size (4 processor):")
plot2 = plot_execution_vs_problem_size(4)
plt.show()

print("\nExecution Time vs Problem Size (9 processor):")
plot3 = plot_execution_vs_problem_size(9)
plt.show()

print("\nExecution Time vs Problem Size (16 processor):")
plot4 = plot_execution_vs_problem_size(16)
plt.show()

print("\nExecution Time vs Problem Size (25 processor):")
plot5 = plot_execution_vs_problem_size(25)
plt.show()

print("\nExecution Time vs Processors (N=1000):")
plot6 = plot_execution_vs_processors(1000)
plt.show()

print("\nExecution Time vs Processors (N=1000):")
plot7 = plot_execution_vs_processors(1000)
plt.show()
```

```
print("\nExecution Time vs Processors (N=100000):")
plot8 = plot_execution_vs_processors(100000)
plt.show()

print("\nParallel Performance Metrics:")
metrics_df = calculate_parallel_metrics()
print(metrics_df.head(10))

print("\nHeatmap for Parallel algorithm:")
plot3 = plot_heatmap("Parallel")
plt.show()

# Additional Analysis: Identify the best algorithm for each problem size
def find_best_algorithm():
    """Find the best algorithm for each problem size and processor count."""
    # Group by N and Processors
    best_algos = df.loc[df.groupby(['N', 'Processors'])['ExecutionTime'].idxmin()]

    # Select relevant columns
    best_algos = best_algos[['N', 'Processors', 'Algorithm', 'ExecutionTime']]

    # Sort by N and Processors
    best_algos = best_algos.sort_values(['N', 'Processors'])

    return best_algos

print("\nBest Algorithm by Problem Size and Processor Count:")
best_algos = find_best_algorithm()
print(best_algos)

# Additional Analysis: Compare regular vs optimized versions
def compare_regular_vs_optimized():
    """Compare regular and optimized versions of algorithms."""
    # Define pairs to compare
    pairs = [
        ('Bit Array With Cyclic', 'Optimized with Cyclic'),
        ('Bit Array With Cannon', 'Optimized with Cannon')
    ]

    for regular, optimized in pairs:
```

```
        plt.figure(figsize=(12, 8))

        # Filter data for single processor
        reg_data = df[(df['Algorithm'] == regular) & (df['Processors'] == 1)]
        opt_data = df[(df['Algorithm'] == optimized) & (df['Processors'] == 1)]

        # Create a merged dataframe
        merged = pd.merge(reg_data, opt_data, on=['N', 'Processors'], suffixes=('_reg', '_opt'))

        # Calculate speedup ratio
        merged['Speedup'] = merged['ExecutionTime_reg'] / merged['ExecutionTime_opt']

        # Plot
        plt.plot(merged['N'], merged['ExecutionTime_reg'], 'b-o', linewidth=2, label=f'{regular}')
        plt.plot(merged['N'], merged['ExecutionTime_opt'], 'r-o', linewidth=2,
label=f'{optimized}')

        plt.title(f'Regular vs Optimized: {regular} comparison')
        plt.xlabel('Problem Size (N)')
        plt.ylabel('Execution Time (seconds)')
        plt.xscale('log')
        plt.yscale('log')
        plt.grid(True, which="both", ls="-")
        plt.legend()

        # Add a second y-axis for speedup ratio
        ax2 = plt.twinx()
        ax2.plot(merged['N'], merged['Speedup'], 'g--', linewidth=2, label='Speedup Ratio')
        ax2.set_ylabel('Speedup Ratio (Regular/Optimized)', color='g')
        ax2.tick_params(axis='y', labelcolor='g')

        plt.tight_layout()
        plt.show()

        print(f"\nSpeedup data for {regular} vs {optimized}:")
        print(merged[['N', 'ExecutionTime_reg', 'ExecutionTime_opt', 'Speedup']])

print("\nComparing Regular vs Optimized Versions:")
compare_regular_vs_optimized()
```

```
# Additional Analysis: Study algorithmic scaling behavior
def analyze_scaling_behavior():
    """Analyze how execution time scales with problem size."""
    plt.figure(figsize=(14, 10))

    # For single processor data
    single_proc = df[df['Processors'] == 1]

    for algo in single_proc['Algorithm'].unique():
        algo_data = single_proc[single_proc['Algorithm'] == algo]

        if len(algo_data) > 3:  # Need at least a few points to analyze scaling
            plt.loglog(algo_data['N'], algo_data['ExecutionTime'], 'o-', linewidth=2, label=algo)

    # Add reference lines for different complexity classes
    n_values = np.logspace(0, 4, 100)

    # Normalize to fit in the plot
    max_y = single_proc['ExecutionTime'].max()
    min_n = single_proc['N'].min()

    plt.title('Algorithm Scaling Behavior')
    plt.xlabel('Problem Size (N)')
    plt.ylabel('Execution Time (seconds)')
    plt.grid(True, which="both", ls="-", alpha=0.5)
    plt.legend(loc='upper left', bbox_to_anchor=(1.05, 1))
    plt.tight_layout()

    plt.show()

print("\nAnalyzing Algorithm Scaling Behavior:")
analyze_scaling_behavior()
```