# Report: Measuring Matrix Sparsity

**Author:** Om Dave (CO22BTECH11006)
**Date:** 14th August 2024

## Introduction

This project focuses on measuring the sparsity of a square matrix using parallel programming techniques in C++. Sparsity is the proportion of zero-valued elements in the matrix, which can significantly impact the performance of various numerical algorithms. Efficient calculation of matrix sparsity using parallel computing methods can lead to improved performance in applications requiring large-scale matrix computations.

## Low-Level Program Design

The program is implemented in C++ using the POSIX Threads library for multithreading. Three primary methods were designed to compute the sparsity in parallel:

- **Chunk:** The matrix is divided into equal segments, and each thread processes a segment.
- **Mixed:** Threads are assigned rows in a round-robin fashion.
- **Dynamic:** Rows are dynamically assigned to threads as they finish their tasks.

Additionally, a hybrid method combining Chunk and Mixed approaches was experimented with.

## Methods

In the Chunk method, the matrix is divided into `N/K` chunks where `N` is the size of the matrix and `K` is the number of threads. Each thread computes the zero-valued elements in its assigned chunk.

**Example**: Consider a 4x4 matrix and 2 threads. The matrix is divided into two chunks:

- The first chunk comprises rows 1 and 2.
- The second chunk comprises rows 3 and 4. Thread 1 is responsible for computing the operations in the first chunk, while thread 2 handles the second chunk.

**Relevant Code Snippet**:

```
void* Compute_chunk(void* arg) {
    ComputeArgs* args = (ComputeArgs*)arg;
    int thread_id = args->thread_id;
    int start = args->start;
    int end = min(start + chunk, n);  // Define 'chunk' size per thread
```

```
    // Adjust the end for the last thread to cover all remaining rows
    if (thread_id == k-1) {
        end = n;
    }

    for (int i = start; i < end; i++) {
        for (int j = 0; j < n; j++) {
            if (A[i][j] == 0) {
                zeros_by_thread[thread_id]++;
            }
        }
    }

    free(args);  // Clean up thread arguments
    pthread_exit(NULL);  // Properly exit thread
}
```

The function `Compute_chunk` is designed for parallel computation of zero values in a matrix. It operates by assigning each thread a specific chunk of rows from the matrix, with the range of rows determined by the thread's ID. As we have divided the matrix into different chunks for each thread, for each thread there would be a starting index of the chunk determined by `(thread->id * chunk_size)` Each thread iterates through its assigned rows and counts the zero values, storing the result in a thread-specific counter.

### 2. Mixed

The Mixed method distributes the matrix rows cyclically among the threads. Each thread computes sparsity for specific rows distributed throughout the matrix.

Example n a 4x4 matrix with 2 threads, the assignment of rows would be as follows: thread 1 computes rows 1 and 3, while thread 2 computes rows 2 and 4. This interleaved assignment ensures that each thread has an equal number of rows to process.

```
void* Compute_mixed(void* arg) {
    ComputeArgs* args = (ComputeArgs*)arg;
    int thread_id = args->thread_id;
    int start = args->start;
    int end = n;

    for (int i = start; i < end; i += k) {
        //computing row i of product matrix
        for (int j = 0; j < n; j++) {
            if (A[i][j] == 0) zeros_by_thread[thread_id]++;
        }
```

```
    }

    free(args);
    pthread_exit(NULL);
}
```

The `Compute_mixed` function implements the Mixed method, which cyclically distributes rows of the matrix among threads. Each thread starts at a specific row and skips a number of rows equal to the total number of threads (`k`). This pattern ensures each thread continuously processes every `k`-th row throughout the matrix, enhancing parallel efficiency by evenly distributing the workload.

### 3. Dynamic

The Dynamic method assigns rows to threads on-the-fly. As a thread completes its task of counting zeros in assigned rows, it picks up the next available set of rows to process.

Example In a 8 by 8 matrix with rowInc = 2, any available thread with take the next 2 set of rows to process and increments the next_row counter. In this fashion, depending on the avaibility, the threads are assigned different sets of rows to process.

```
atomic<int> C(0); // Atomic Shared counter to keep track of number of
rows allocated
void* Compute_Dynamic(void* arg) {
    ComputeArgs* args = (ComputeArgs*)arg;
    int thread_id = args->thread_id;

    while (C <= n) {
        int start = C.fetch_add(rowInc); //Atomically returns the
previous value of C counter and adds rowInc to it, basically gives the
starting index for this to calculate
        int end = min(start + rowInc, n);

        for (int i = start; i < end; i++) {
            //computing row i of product matrix
            for (int j = 0; j < n; j++) {
                if (A[i][j] == 0) zeros_by_thread[thread_id]++;
            }
        }
    }

    free(args);
    pthread_exit(NULL);
}
```

The `Compute_Dynamic` function utilizes an atomic counter `C` to dynamically allocate rows to threads. Each thread accesses `C` to obtain the start index of the next set of rows (`rowInc`), then processes these rows by iterating over each element to count zeros.The atomic counter is incremented using `fetch_and_add()` operation which ensures that there are no race conditions amongst the threads and each thread takes the next free set of rows. This ensures threads are continuously engaged by pulling new tasks as they complete their current assignments, optimizing parallel efficiency and minimizing downtime across all active threads.

### 4. Mixed-Chunk

The Mixed-Chunk method is a hybrid approach that first divides the matrix into small chunks and then distributes these chunks cyclically among the threads. This method aims to balance load dynamically while minimizing synchronization overhead.

Example: Divide the A matrix into chunks of rows, where each chunk contains rowInc rows and then distribute these chunks in round-robin fashion among a subset of threads. In this way each thread will be responsible for a set of chunks of rows of matrix.

```
// Thread function to count number of zeros
void* Compute_Mixed_Chunk(void* arg) {
    ComputeArgs* args = (ComputeArgs*)arg;
    int thread_id = args->thread_id;
    int start = args->start;
    int end = n;

    for (int p = start; p < end; p += k * rowInc) {
        for (int i = p; i < min(p + rowInc, n); i++) {
            for (int j = 0; j < n; j++) {
                if (A[i][j] == 0) zeros_by_thread[thread_id]++;
            }
        }
    }

    free(args);
    pthread_exit(NULL);
}
```

The `Compute_Mixed_Chunk` function implements the Mixed-Chunk method, which integrates chunk and mixed strategies to distribute matrix rows among threads. Each thread calculates zeros in a specific segment that spans the entire matrix. Starting from a designated row, each thread skips `k * rowInc` rows to ensure a balanced and staggered distribution of rows across multiple threads. This approach reduces synchronization overhead and evenly spreads the workload, improving parallel efficiency while ensuring each thread remains continuously engaged by processing successive blocks of rows.

# Performance Analysis

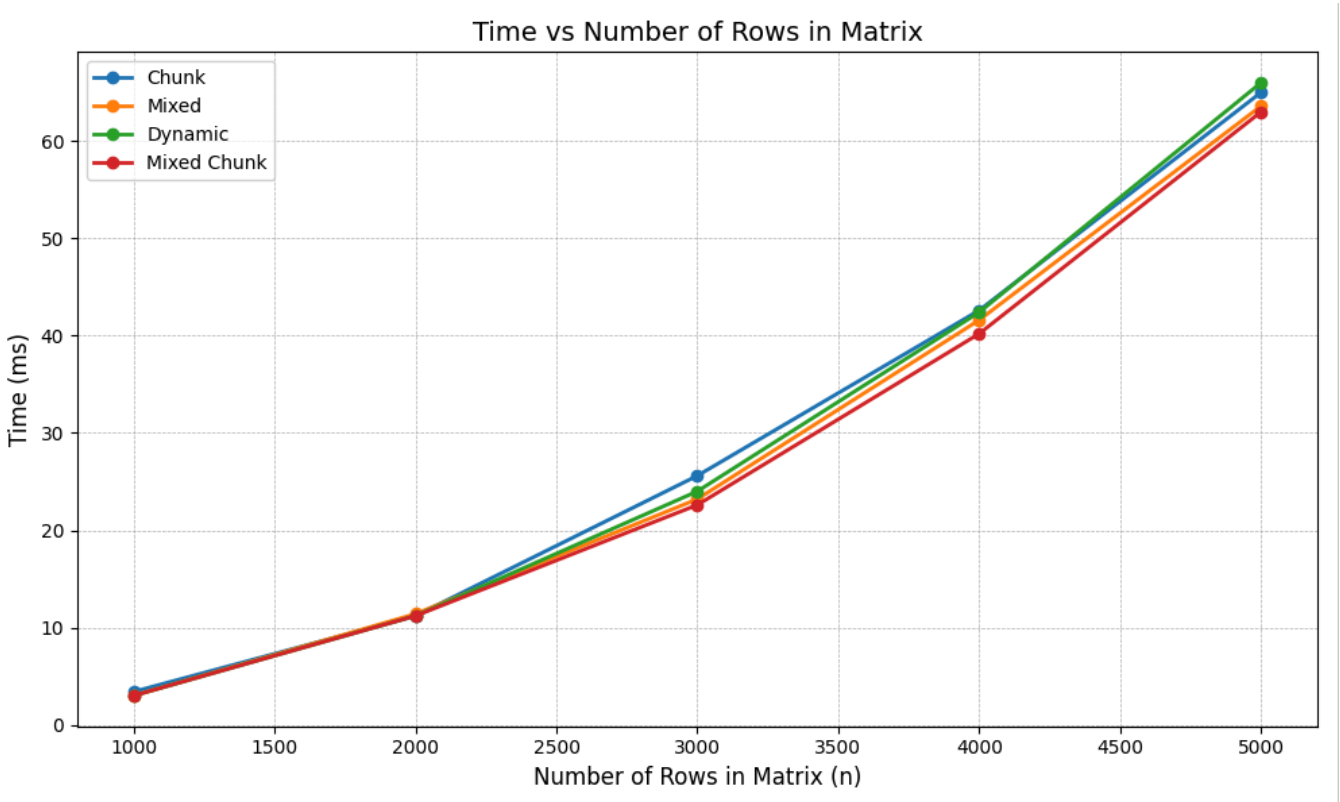
## Experiment 1: Time vs. Size, N

### Introduction

This experiment investigates the impact of matrix size on the time required to compute sparsity using the 4 parallel methods: Chunk, Mixed,Mixed-Chunk and Dynamic. The matrix sizes tested range from 1000x1000 to 5000x5000 elements, with sparsity held constant at 40% and the number of threads fixed at 16. This setup helps to determine the scalability and efficiency of each method as the computational load increases.

### Experiment 1: Total Number of Zero-Valued Elements by Matrix Size

| Matrix Size | Chunk Method | Mixed Method | Dynamic Method | Mixed-Chunk Method |
|---|---|---|---|---|
| 1000x1000 | 400,000 | 400,000 | 400,000 | 400,000 |
| 2000x2000 | 1,600,000 | 1,600,000 | 1,600,000 | 1,600,000 |
| 3000x3000 | 3,600,000 | 3,600,000 | 3,600,000 | 3,600,000 |
| 4000x4000 | 6,400,000 | 6,400,000 | 6,400,000 | 6,400,000 |
| 5000x5000 | 10,000,000 | 10,000,000 | 10,000,000 | 10,000,000 |




### Observations

The graph shows the computation time for each parallel method as the matrix size increases from 1000x1000 to 5000x5000. All methods (Chunk, Mixed, Dynamic, and Mixed-Chunk) display a increase in computation time as the matrix size grows. The computation times for the four methods remain closely packed throughout the range, with Dynamic and Mixed-Chunk methods slightly outperforming the Chunk and Mixed methods, especially at larger sizes.

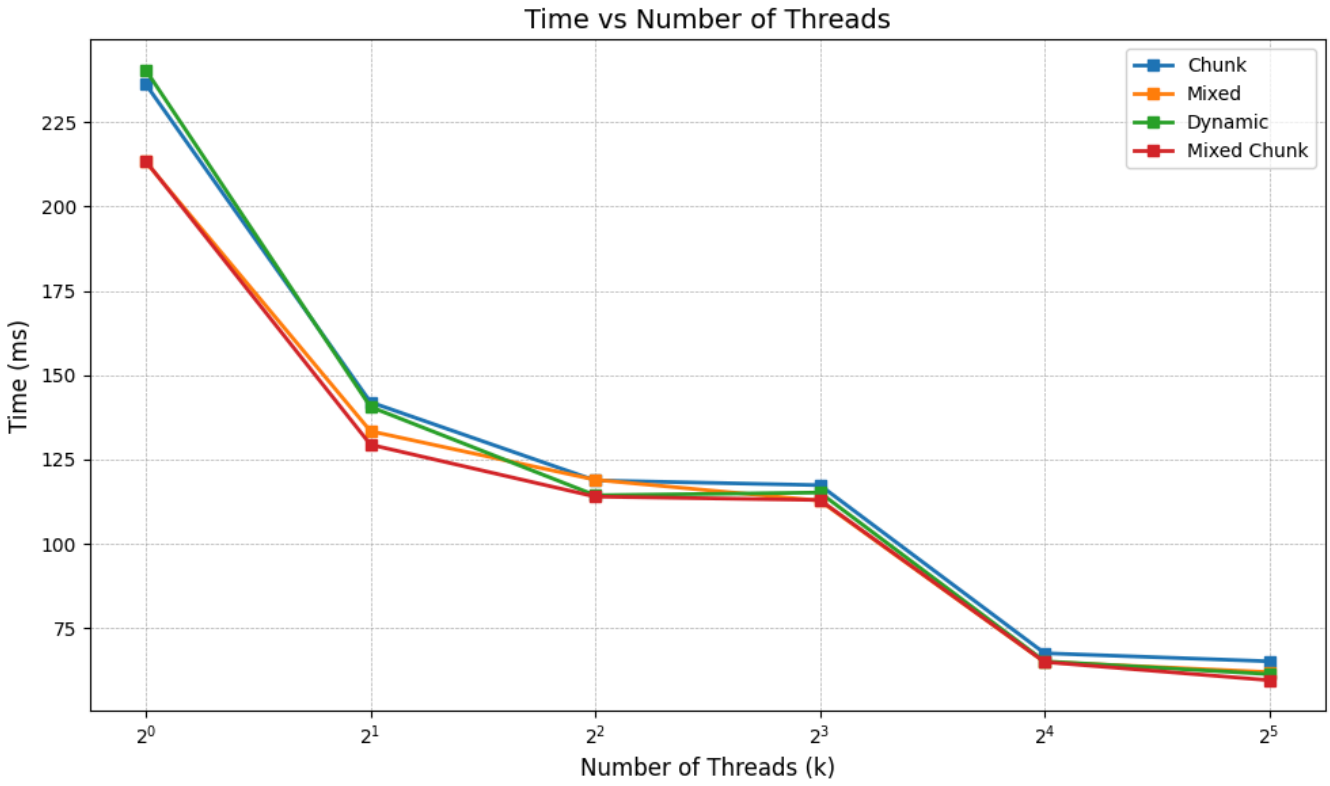
**Reasons**

- **Increase in Computation Time:** The increase in time with size for all methods is expected as the total number of matrix elements, and hence the computational load, increases with the size of the matrix.
- **Close Performance:** The close performance across methods can be attributed to the fact that all methods effectively parallelize the workload. However, slight differences in performance indicate varying efficiencies in workload distribution and overhead management.
- **Dynamic and Mixed-Chunk Advantage:** The slightly better performance of the Dynamic and Mixed-Chunk methods at larger sizes suggests that these methods may be better at dynamically balancing the workload among threads, minimizing idle time and efficiently handling varying amounts of work per thread.

## Experiment 2: Time vs. Number of Threads, K

**Introduction**

In this experiment, the focus shifts to the effect of varying the number of threads on the computation time for a fixed matrix size of 5000x5000. Thread counts are adjusted through the powers of two, ranging from 1 to 32. This experiment aims to evaluate the concurrency efficiency of each method, identifying how well they leverage multi-threading to speed up the sparsity computation.




**Observations**

The graph shows a significant decrease in computation time as the number of threads increases from 1 to 32. All methods (Chunk, Mixed, Dynamic, and Mixed-Chunk) demonstrate a sharp reduction in time when moving from a single thread to two threads, with further reductions as the number of threads doubles. Beyond 16 threads, the reduction in computation time plateaus, indicating a diminishing return on adding more threads.

**Reasons**

- **Sharp Reduction Up to 16 Threads:** The sharp reduction in computation time with increasing threads up to 16 can be attributed to more efficient parallel processing of the matrix. As each thread can handle a portion of the matrix independently, increasing the thread count significantly reduces the workload per thread, leading to faster overall computation times.
- **Diminishing Returns Beyond 16 Threads:** The plateauing effect observed beyond 16 threads suggests that there are limits to the benefits of adding more threads. This could be due to several factors:
    - **Thread Overhead:** As the number of threads increases, the overhead associated with managing these threads can offset the gains from parallel processing.
    - **Hardware Limitations:** The hardware might not effectively support more than a certain number of threads, leading to underutilization of added threads.

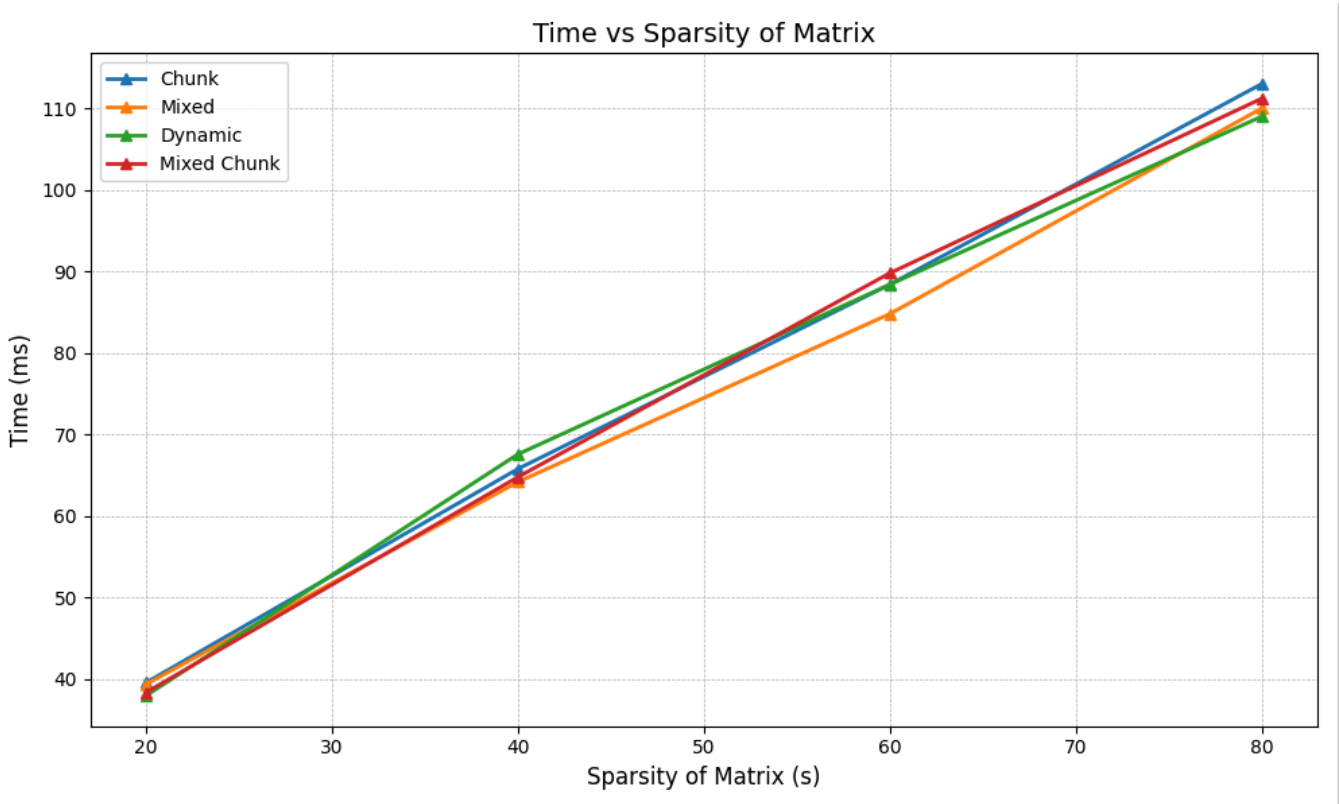
## Experiment 3: Time vs. Sparsity, S

**Introduction**

This experiment explores how the level of sparsity (ranging from 20% to 80%) affects the computation time. The matrix size is held constant at 5000x5000, with 16 threads participating in the computation. By altering the sparsity, we can observe how the density of nonzero elements influences the processing time across different parallel computing techniques.

**Experiment 3: Total Number of Zero-Valued Elements by Sparsity Value**

| Sparsity (%) | Chunk Method | Mixed Method | Dynamic Method | Mixed-Chunk Method |
|---|---|---|---|---|
| 20% | 5,000,000 | 5,000,000 | 5,000,000 | 5,000,000 |
| 40% | 10,000,000 | 10,000,000 | 10,000,000 | 10,000,000 |
| 60% | 15,000,000 | 15,000,000 | 15,000,000 | 15,000,000 |
| 80% | 20,000,000 | 20,000,000 | 20,000,000 | 20,000,000 |




**Observations**

The graph illustrates that the computation time for all four methods (Chunk, Mixed, Dynamic, and Mixed-Chunk) increases as the sparsity of the matrix rises from 20% to 80%. The trends for each method are closely aligned, indicating that the performance remains consistent across different levels of sparsity. Notably, the Dynamic and Mixed-Chunk methods show slightly better performance compared to the Chunk and Mixed methods, but the overall difference is minimal.

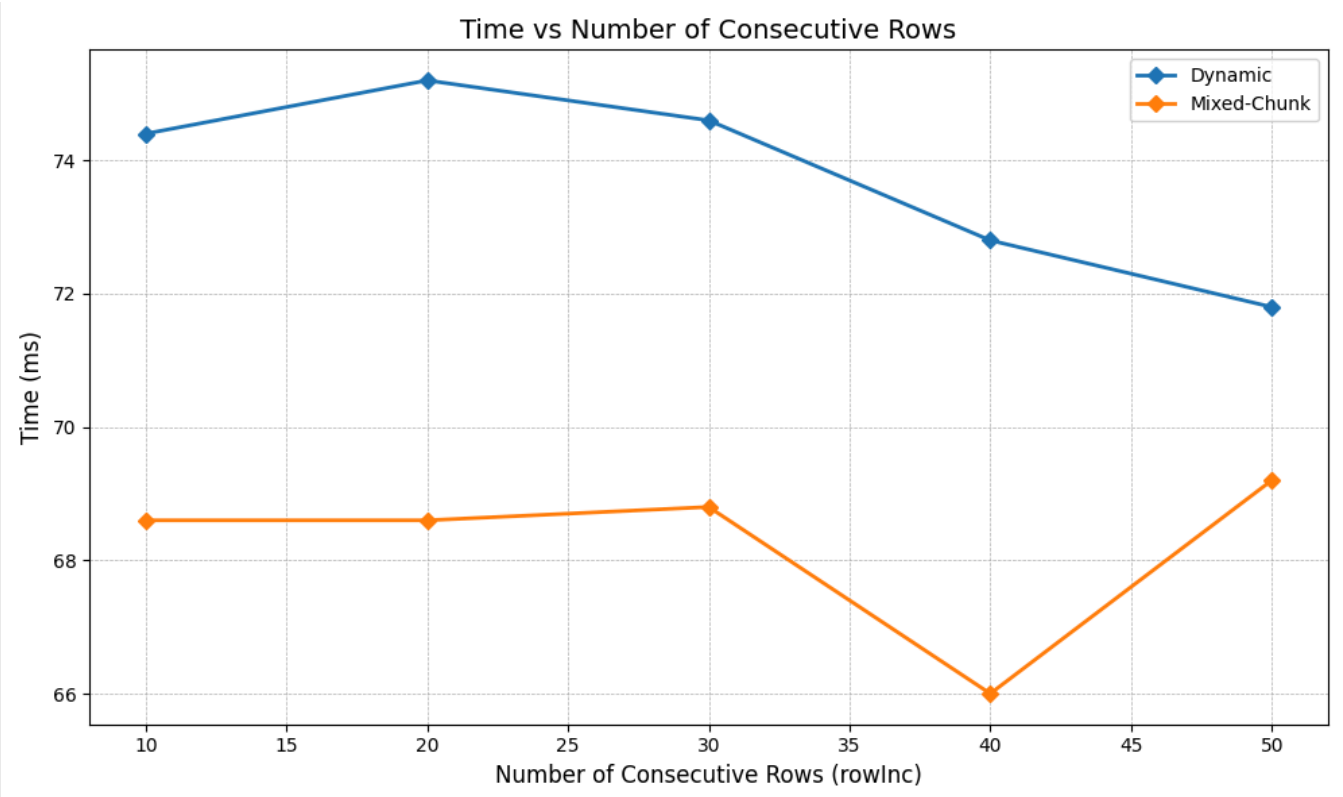
**Reasons**

- **Increase in Computation Time:** The increase in computation time as sparsity rises can be attributed to the higher number of zero elements in the matrix. As the number of zeros increases, threads must perform more increment operations to account for these zero elements, leading to higher computation times across all methods.
- **Consistent Performance Across Methods:** The similar performance across different methods suggests that the presence of more zeros uniformly affects all methods. However, the Dynamic method might have a slight edge due to its ability to dynamically allocate rows, which helps in managing the increased workload caused by a higher number of zeros.

## Experiment 4: Time vs. Row Increment, rowInc

**Introduction**

Focusing solely on the Dynamic method, this experiment varies the row increment values (10, 20, 30, 40, 50) to study their effect on the computation time. The matrix size is fixed at 5000x5000, sparsity at 40%, and 16 threads are used. This setup is intended to pinpoint the optimal row increment that maximizes efficiency in dynamic row assignment during matrix sparsity calculations.




**Observations**

The graph displays the computation time for the Dynamic and Mixed-Chunk methods as the row increment (`rowInc`) varies from 10 to 50. The Dynamic method shows a consistent decrease

in computation time as the number of consecutive rows increases, indicating improved performance with larger chunks of rows. In contrast, the Mixed-Chunk method exhibits a relatively stable performance with a slight increase in time until `rowInc` 40, followed by a sharp increase at `rowInc` 50.

**Reasons**

- **Dynamic Method Decrease:** The decreasing trend in the Dynamic method can be attributed to the reduced overhead of task management. As `rowInc` increases, each thread deals with larger blocks of the matrix at a time, reducing the frequency of synchronization and task allocation overhead. This leads to more efficient utilization of threads.
- **Stability in Mixed-Chunk:** The relatively stable performance of the Mixed-Chunk method for lower row increments suggests that this method effectively balances the load among threads, even when dealing with smaller chunks. This is likely due to the inherent efficiency of the mixed assignment within each chunk.

## Conclusion

The series of experiments conducted to measure matrix sparsity using parallel computing techniques in C++ have demonstrated significant insights into the performance and scalability of different methods under varying conditions. The Dynamic and Mixed-Chunk methods generally showed superior performance, particularly in handling larger matrix sizes and higher thread counts, highlighting their potential in real-world applications that demand efficient parallel computations.