CS5300 - Parallel & Concurrent Programming:
Autumn 2024
Programming Assignment 1: Measuring Matrix Sparsity

Name: Dheeraj M
Role Number: EE21BTECH11015

Goal:- This assignment aims to count the number of zero-valued elements in a square matrix
through static and dynamic mechanisms in C++.

1 Chunk technique :

1.1 Description:

- The Chunk method divides the matrix into equal-sized segments called chunks. Each chunk is assigned to a different thread. This approach leverages parallel processing to speed up the counting of zero-valued elements.

1.2 Low-Level Design: The low-level design section describes the detailed implementation of your program, including how different components interact at the code level. Here's an elaboration based on your Chunk method code:

1.2.1 Class Definition:

- Class `sparseMatrix`:
  - Attributes:
    - `int N, S, K, rowInc`: Matrix dimensions and parameters.
    - `vector<vector<int>> vectorMatrix`: 2D vector to store the matrix data.
  - Methods:
    - While the constructor is commented out, a constructor could initialize the matrix or parameters if needed.

1.2.2 Function Definitions

- **`PopulateMatrix()` Function:**
  - **Purpose:** To read matrix data from a file and populate the `sparseMatrix` object.
  - **Parameters:**
    - `ifstream& f`: Input file stream for reading matrix data.
    - `sparseMatrix* obj`: Pointer to the `sparseMatrix` object to be populated.
  - **Implementation:**
    - Reads the matrix dimensions and parameters from the first line.
    - Resizes the matrix to the specified dimensions.
    - Reads subsequent lines to fill the matrix with data.

        - Includes error handling for mismatched row sizes and total row count.

- **`SparseCounting()` Function:**
    - **Purpose:** To count zero-valued elements in a specified segment of rows in the matrix.
    - **Parameters:**
        - `int start, int end`: Indices of the rows to be processed.
        - `sparseMatrix* mat`: Pointer to the matrix object.
        - `int* totalSum`: Pointer to an integer to accumulate the total count of zeros.
        - `mutex* mtx`: Pointer to a mutex for thread synchronization.
        - `vector<int>& threadCount`: Vector to store zero counts for each thread.
        - `int threadIndex`: Index of the current thread.
    - **Implementation:**
        - Iterates over rows from `start` to `end`.
        - Uses `std::count` to count zeros in each row.
        - Locks and updates the `totalSum` and `threadCount` with mutex to ensure thread safety.
        - Logs debug information for tracking the process.
- **`chunk()` Function:**
    - **Purpose:** To divide the matrix into chunks and process each chunk using multiple threads.
    - **Parameters:**
        - `sparseMatrix* mat`: Pointer to the matrix object.
        - `vector<thread>& ths`: Vector to hold thread objects.
        - `ofstream& outFile`: Output stream to log results.
    - **Implementation:**
        - Calculates the number of rows per chunk (`p = N / K`).
        - Initializes thread management and synchronization tools.
        - Creates threads to process each chunk of rows.
        - Uses a mutex to synchronize access to shared data (`totalSum`).
        - Measures and averages the execution time over multiple iterations.
        - Logs results to output files.
- **`processingFiles()` Function:**
    - **Purpose:** To handle file reading, matrix processing, and logging.
    - **Parameters:**
        - `const string& filePath`: Path to the input file.
        - `ofstream& outFile, outFile2, outFile3`: Output streams for different logging purposes.
    - **Implementation:**
        - Opens and reads the input file.
        - Populates the matrix using `PopulateMatrix`.
        - Calls `chunk` function to process the matrix.
        - Adjusts parameters (`K` and `rowInc`) and repeats processing to gather performance data.

- **`main` Function:**
  - **Purpose:** To execute the program and process multiple files.
  - **Parameters:** None
  - **Implementation:**
    - Defines the input folder and output file paths.
    - Opens the directory containing input files.
    - Processes each file using `processingFiles`.
    - Manages output files for different configurations and parameters.

1.2.3 Thread Management

- **Thread Creation:**
  - Threads are created using `std::thread` and assigned to process specific chunks of the matrix.
  - Lambda functions are used to pass parameters to the `SparseCounting` function.
- **Thread Synchronization:**
  - Mutex: A `std::mutex` is used to protect shared resources (`totalSum` and `threadCount`) from concurrent access.
  - Locking: Mutex is locked before updating shared data and unlocked afterward to prevent race conditions.
- **Thread Joining:**
  - Ensures that the main thread waits for all worker threads to complete using `th.join()`.

1.2.4 Performance Measurement

- **Timing:**
  - High-Resolution Clock: `std::chrono::high_resolution_clock` is used to measure the time taken for processing.
  - Averaging: Execution time is averaged over multiple iterations to provide a more accurate performance measure.
- **Output Logging:**
  - Results: Time taken and zero counts are logged to output files.
  - Configuration Details: The number of threads and row increments are logged to track their impact on performance.

2 Mixed technique :

2.1 Description:

- The Mixed method for counting zero-valued elements in a matrix involves splitting the matrix rows among multiple threads, each of which processes a subset of the rows to count zero elements. This method combines multithreading with careful synchronization to ensure accurate and efficient results.

2.2 Low-Level Design

2.2.1 Class Defination

- "Same as Chunk method"

2.2.2 Function Defination

- **`SparseCounting()` Function:**
  - **Purpose:** Counts zero elements in a specified list of rows.
  - **Parameters:**
    - `vector<int> rowIndexList:` List of row indices to process.
    - `sparseMatrix* mat:` Pointer to the matrix object.
    - `int* totalSum:` Pointer to the total sum of zeros.
    - `mutex* mtx:` Mutex for synchronization.
    - `vector<int>& threadCount:` Vector to store counts for each thread.
    - `int threadIndex:` Index of the current thread.
  - **Steps:**
    - Count zero elements in the specified rows.
    - Lock the mutex, update the total zero count, and store the count for the current thread.
    - Unlock the mutex.
- **`mixed()` Function**
  - **Purpose:** The primary purpose of the `mixed` function is to efficiently count the number of zero-valued elements in the matrix by leveraging multithreading. It distributes the rows of the matrix among threads and aggregates the results while measuring execution time.
  - **Parameters:**
    - `sparseMatrix* mat`: Pointer to the `sparseMatrix` object containing the matrix data.
    - `vector<thread>& ths`: Vector to store the created threads.
    - `ofstream& outFile`: Output file stream for writing results.
  - **Steps:**
    - **Initialization**
      - `totalSum`: Initialized to 0; will hold the total count of zero elements.
      - `num_threads`: Set to `mat->K`, the number of threads to use.
      - `rowIndexList`: Vector to temporarily hold indices of rows assigned to a thread.

- `mtx`: Mutex object for synchronizing access to shared variables.
- `threadCount`: Vector to store the count of zero elements processed by each thread.
- `time_avg`: Variable to accumulate average execution time.

- **Row Distribution and Thread Creation**
  - `rowIndexList.clear()`: Clears the list of row indices for the current iteration.
  - Distribute Rows:For each thread (`for (int i = 1; i <= num_threads; i++)`):
    - Initialize `rowIndex` for the current thread.
    - Update `rowIndexList` with rows assigned to this thread based on the row distribution pattern.
    - Create a lambda function to count zeros in the rows of `rowIndexList` and start a thread to execute this lambda function.
    - The lambda function captures `rowIndexList`, `mat`, `totalSum`, `mtx`, `threadCount`, and the thread index `i`.

2.2.3 Thread Management

- `ths.push_back(thread(...))`: Adds the created thread to the `ths` vector.
- `rowIndexList.clear()`: Clears the list after creating the thread to ensure that the next thread starts with a fresh list.

2.2.4 Key Aspects:

- **Thread Distribution:** Rows are distributed among threads in a way that each thread processes a subset of the matrix rows. The distribution is based on the number of threads (`mat->K`) and a pattern that ensures all rows are covered.
- **Lambda Functions:** Lambda functions are used to encapsulate the row processing logic and are executed by the threads. This allows for a clean and concise way to pass parameters to the threads.
- **Synchronization:** Mutex (`mtx`) is used to synchronize access to the `totalSum` variable to prevent race conditions when multiple threads update this shared variable.
- **Performance Measurement:** The function measures execution time over multiple iterations to average out variability and provide a more stable measure of performance.

3 Dynamic technique :

3.1 Description:

- The dynamic technique dynamically allocates the rows of a sparse matrix to threads for counting the number of zeros. The rows are assigned in chunks defined by `rowInc` to each thread, which increments a shared counter to determine the set of rows it processes. The number of threads (`K`) and the chunk size (`rowInc`) are parameters that can be varied to evaluate performance.

3.2 Low-Level Design:

3.2.1 Class Definitions

- "Same as Chunk method"

3.2.2 Function Definitions

- **`SparseCounting()` Function:**
  - **Purpose**
    - The primary goal of `SparseCounting` is to count zeros in assigned chunks of the matrix and update both the total zero count and the individual thread's zero count.
  - **Arguments:**
    - `mat`: Pointer to the `sparseMatrix` object.
    - `totalSum`: Pointer to the shared variable holding the total count of zeros.
    - `mtx`: Mutex for synchronizing access to shared variables.
    - `start`: Pointer to the shared variable indicating the starting row for the next chunk.
    - `threadCount`: Vector to store the count of zeros each thread finds.
    - `threadIndex`: The index of the current thread.
  - **Steps:**
    - Dynamic Row Assignment:
      - The function locks a mutex and assigns a chunk of rows (`rowInc` rows) to the current thread.
      - The starting row is stored in `s`, and the end row is calculated as the minimum of (`s + rowInc`) and `N`.
      - `start` is updated to point to the next chunk's starting row.
    - Zero Counting:
      - The function iterates over the assigned rows and counts the number of zeros in each row.
      - The result is added to both the thread's individual count (`threadCount[threadIndex]`) and the global `totalSum`.
    - Mutex Usage:
      - The mutex ensures that updates to `start`, `totalSum`, and `threadCount` are thread-safe.
- **`Dynamic()` Function:**
  - **Purpose:**

        - The `dynamic` function is designed to manage and execute the parallel counting of zero-valued elements in a sparse matrix using multiple threads. It dynamically allocates rows to threads and measures performance.
    - **Arguments:**
        - `mat`: Pointer to the `sparseMatrix` object.
        - `ths`: Reference to a vector of threads.
        - `outFile`: Output file stream to log results.
    - **Steps:**
        - **Initialization:**
            - The function initializes variables including `totalSum` (to store the total count of zeros), `threadCount` (a vector to track zeros counted by each thread), and a mutex (`mtx`) for synchronizing access to shared data.
        - Thread Creation and Execution:
            - **Thread Creation:**
                - The function creates a specified number of threads (`mat->K`). Each thread executes the `SparseCounting` function, which processes chunks of rows from the matrix.
                - Threads are created with a lambda function that captures the matrix (`mat`), the shared variables (`totalSum`, `mtx`, `start`), the index of the thread (`i`), and the `threadCount` vector.
            - **Thread Execution:**
                - Each thread runs concurrently and counts zeros in its assigned rows. The `SparseCounting` function is responsible for dynamically fetching and processing rows based on the current `start` index and `rowInc`.
        - **Synchronization and Joining:**
            - After starting all threads, the function waits for each thread to complete its execution using `join()`. This ensures that all threads finish processing before moving on.

3.2.3 Thread Management

- `ths.push_back(thread(...))`: Adds each created thread to the `ths` vector. This vector keeps track of all threads that have been spawned, allowing for their management and synchronization. Each thread performs the zero counting operation on different chunks of the matrix.
- `rowIndexList.clear()`: Although not explicitly used in the provided code, this operation is generally used to reset the list of row indices after thread creation. Ensures that subsequent threads start with an updated or fresh list, preventing conflicts or reprocessing of rows.

3.2.4 Key Aspects

- **Thread Distribution:** The matrix rows are dynamically allocated to threads based on the `rowInc` parameter, which specifies the number of rows each thread processes in one

iteration. This method ensures that rows are distributed among threads efficiently and that all rows are covered without overlap.

- **Lambda Functions:** Lambda functions are employed to define the work each thread performs. These functions capture necessary variables (e.g., matrix pointer, shared variables) and execute the `SparseCounting` function, allowing for a clean and flexible way to pass parameters and encapsulate thread-specific logic.
- **Synchronization:** A mutex (`mtx`) is utilized to manage concurrent access to shared variables, such as the `totalSum` and the `start` index. This prevents race conditions and ensures that updates to these shared resources are performed safely, maintaining data consistency.
- **Performance Measurement:** The function measures the time taken to execute the zero counting operation across multiple iterations to average out any variability in execution time. This approach provides a reliable assessment of the performance of the dynamic threading technique and helps identify optimal configurations for thread usage and row allocation.

4 Experiments:

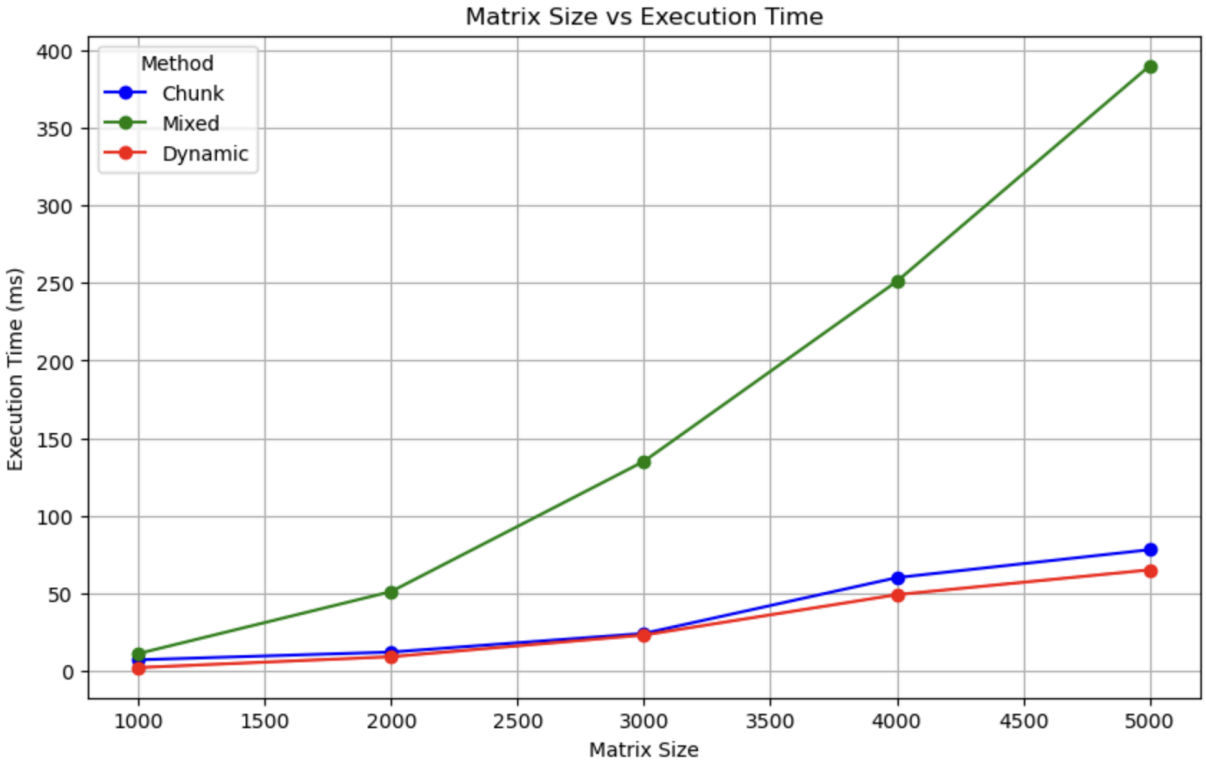
4.1 Time vs Size, N :

- Sparsity of the matrix(S): 40%
- Number of threads(K): 16
- rowInc(Dynamic): 50

The table-1 describes the number of Zero elements for a particular matrix dimension.

	Matrix Size	Total Number of Zeros
0	5000	10000000
1	3000	3600000
2	4000	6400000
3	2000	1600000
4	1000	400000

Table-1 Zero-Valued Elements by Matrix Size

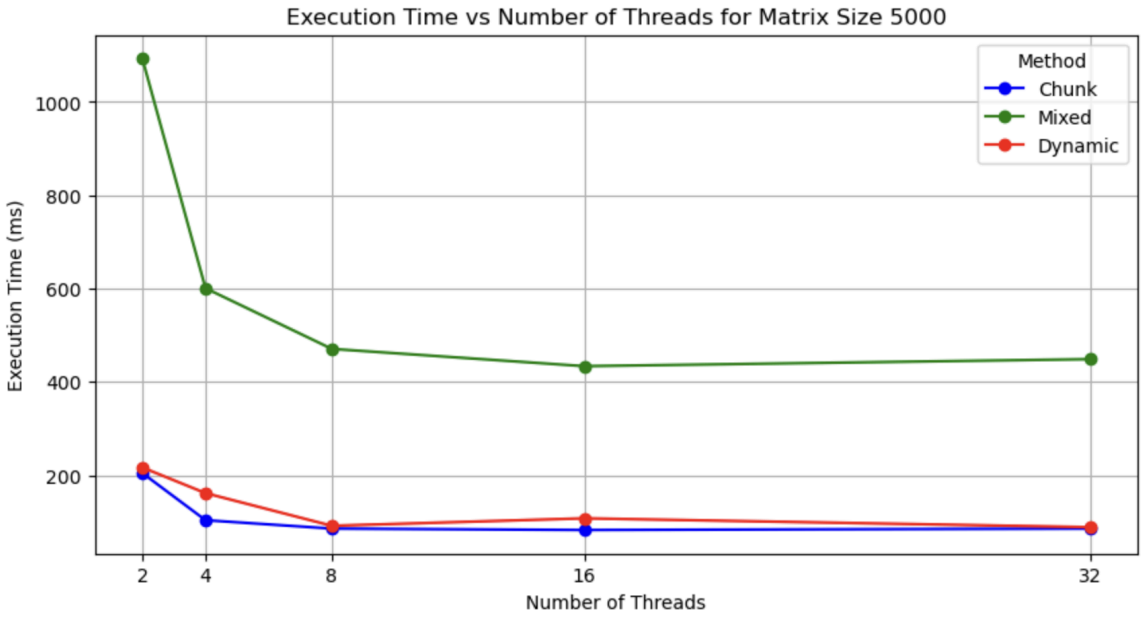


4.1.1 Observation

- **Overall Trend**:
  - All methods exhibit an increase in execution time as matrix size grows, which is expected since larger matrix sizes translate to a heavier workload for the threads.
- **Chunk Method (Blue Line)**:
  - This method performs well, with execution time increasing gradually and showing only minor changes between different matrix sizes. This demonstrates the method's ability to efficiently handle larger matrix sizes.
- **Mixed Method (Green Line)**:
  - The Mixed method performs poorly, with execution time spiking significantly when the matrix size increases from 1000 to 2000. In contrast, the other two methods maintain much lower execution times for the same increase.
  - As the matrix size continues to grow, the execution time for the Mixed method escalates dramatically, highlighting its inefficiency for larger matrix sizes.
- **Dynamic Method (Red Line)**:
  - The Dynamic method performs similarly to the Chunk method. But being the fasted among all.
- **Conclusion**: The Chunk and Dynamic methods are the fastest, with the Dynamic method slightly outperforming the Chunk, while the Mixed method is the slowest of the three.

4.2 Time vs. Number of threads, K:

- Sparsity of the matrix(S): 40%
- Matrix Size: 5000
- rowInc(Dynamic): 50



4.2.1 Observations:

- **Overall Trend:**
  - All methods show a decrease in execution time as the number of threads increases, which is expected as more threads typically lead to better parallelism and faster execution.
- **Chunk Method (Blue Line):**
  - The Chunk method has a relatively low execution time across all thread counts, remaining fairly consistent as the number of threads increases.
  - It shows a slight reduction in execution time from 2 to 4 threads, after which it stabilizes.
- **Mixed Method (Green Line):**
  - The Mixed method starts with a much higher execution time when using 2 threads.
  - As the number of threads increases, the execution time drops significantly, particularly from 2 to 4 threads, and then gradually decreases.
  - However, it still remains higher than the other methods, suggesting that the Mixed method might be less efficient for this particular matrix size and configuration.
- **Dynamic Method (Red Line):**
  - The Dynamic method starts with a higher execution time than the Chunk method but lower than the Mixed method.
  - Similar to the Chunk method, it shows a decrease in execution time as the thread count increases, and the time stabilizes after 4 threads.
- **Optimal Thread Count:**

- ➢ For the Chunk and Dynamic methods, using 8 or more threads provides a stable and low execution time, indicating that increasing the number of threads may not yield significant performance improvements beyond this point.
- ➢ The Mixed method benefits the most from increasing the number of threads, but its execution time remains higher than the other two methods across all thread counts.

❖ **Conclusion:** The Chunk and Dynamic methods are more efficient for this matrix size, with the Chunk method having the most consistent performance across different thread counts. The Mixed method appears less efficient, especially at lower thread counts.

4.3 Time vs. Sparsity, S:

- ❖ rowInc(Dynamic): 50
- ❖ Matrix Size: 5000
- ❖ Number of threads(K): 16

	Sparsity(%)	Number of Zeros
0	4000	20000000
1	2000	10000000
2	1000	5000000
3	3000	15000000

Table-2 Zero-Valued Elements by Matrix Size

4.3.1 Observations:

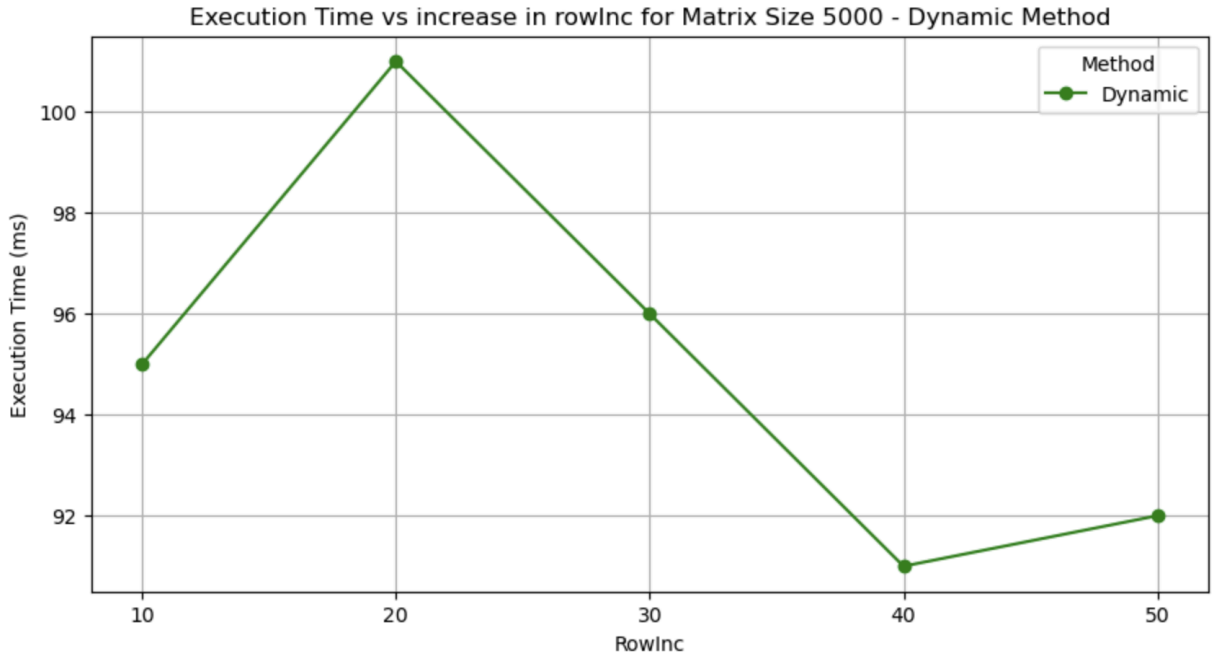
- **Execution Time for Chunk and Mixed Methods:**
  - The Chunk method (blue line) and Mixed method (green line) have relatively lower execution times compared to the Dynamic method (red line).
  - The execution time for the Chunk method remains relatively constant across different levels of sparsity, with only a slight increase.
  - The Mixed method shows an increase in execution time with increasing sparsity, reaching a peak at a sparsity level of 2000 (or 40%) and then slightly decreasing.
- **Execution Time for the Dynamic Method:**
  - The Dynamic method shows a significantly higher execution time compared to the Chunk and Mixed methods.
  - The execution time increases initially, peaks at a sparsity level of 2000 (40%), and then gradually decreases as sparsity increases further.
  - This method's performance varies more across different sparsity levels compared to the other two methods.
- **Comparison Across Methods:**
  - **The Chunk method consistently has the lowest execution time across all sparsity levels.**
  - **The Mixed method has a higher execution time than the Chunk method but lower than the Dynamic method.**
  - **The Dynamic method has the highest execution time, especially noticeable at lower sparsity levels.**
- **Impact of Sparsity on Execution Time:**
  - Sparsity represents the proportion of zero-valued elements in the matrix. As sparsity increases, the number of zero-valued elements increases.
  - Higher sparsity can lead to increased execution time due to the larger number of elements that need to be processed and counted.
  - The Chunk and Mixed methods appear to handle increasing sparsity more efficiently than the Dynamic method, as indicated by the smaller increase in execution time.
- **Chunk Method Efficiency:**
  - The Chunk method divides the matrix into fixed chunks, with each thread responsible for a specific set of rows. This approach minimizes synchronization and communication overhead, leading to more efficient parallel processing.
  - The consistent performance across varying sparsity levels suggests that this method is robust and efficiently utilizes the threads and available resources.
- **Mixed Method Performance:**
  - The Mixed method distributes rows among threads in a staggered manner. This method is slightly less efficient than the Chunk method but still performs relatively well.
  - The slight increase in execution time with increasing sparsity indicates some overhead in managing the distribution of rows among threads, especially at higher sparsity levels.
- **Dynamic Method Challenges:**
  - The Dynamic method allocates rows dynamically to threads, which introduces flexibility but also adds overhead due to the need for synchronization and coordination among threads.
  - The higher execution time, especially at lower sparsity levels, suggests that the overhead associated with dynamic allocation outweighs the benefits in this scenario.

- The decrease in execution time at higher sparsity levels may be due to fewer non-zero elements requiring less synchronization as threads complete their work more independently.

- **Conclusion:**
  - The Chunk method is the most efficient across all levels of sparsity, maintaining low execution times.
  - The Mixed method performs well but has a slight increase in execution time with higher sparsity levels.
  - The Dynamic method has the highest execution times due to the overhead of dynamic allocation and synchronization, making it less suitable for matrices with lower sparsity.
  - Overall, the Chunk method is recommended for its robustness and efficiency in handling varying sparsity levels in parallel matrix processing.

4.4 Time vs. row Increment, rowInc:

- Sparsity of the matrix(S): 40%
- Matrix Size: 5000
- Number of threads(K): 16



4.4.1 Observations:

- **Initial Increase in Execution Time (rowInc = 10 to 20):**
  - The execution time increases significantly as the **rowInc** value increases from 10 to 20. This indicates that allocating more rows per thread initially increases the overall time to compute the sparsity.
- **Peak Execution Time at rowInc = 20:**
  - The peak execution time is observed at a **rowInc** of 20. This suggests that at this point, the balance between the workload per thread and the overhead of thread management is not optimal.
- **Decrease in Execution Time (rowInc = 20 to 40):**

- As **rowInc** increases from 20 to 40, there is a notable decrease in execution time. This implies that increasing the number of rows each thread processes allows for more efficient use of resources and reduces overhead.

- **Slight Increase at rowInc = 50:**
  - There is a slight increase in execution time when the **rowInc** is increased from 40 to 50. This may suggest that the additional workload per thread begins to introduce inefficiencies beyond a certain point, possibly due to increased contention for shared resources or an imbalance in workload distribution or an increase in the processing time of each thread.
- **Conclusion:** The graph demonstrates the effects of varying the rowInc on the execution time of the dynamic allocation method. The analysis reveals that there is an optimal range for rowInc (around 40), where the balance between workload and synchronization overhead is most effective. Beyond this point, increasing the workload per thread can lead to inefficiencies and longer execution times.

5 Conclusion:

the Chunk method emerges as the most effective and efficient approach for parallel matrix processing. It consistently provides the lowest execution times across various matrix sizes, thread counts, and levels of sparsity. While the Mixed method shows improvement with increased thread count, it generally lags in performance due to its variability and higher execution times as matrix size grows. The Dynamic method, despite its flexibility, suffers from overheads related to synchronization and dynamic allocation, making it less optimal overall. Therefore, due to its balance of efficiency, robustness, and simplicity, the Chunk method is the preferred choice for handling parallel matrix processing tasks.