CS5300 - Parallel & Concurrent Programming:
Autumn 2024
Programming Assignment2:Measuring Matrix Sparsity using OpenMP

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++ using OpenMP and compare its performance with that of threads implemented in the previous assignment (Programming Assignment 1).

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.
- ❖ **`chunk()` Function:**
    - ➢ **Purpose:** Processes the matrix in parallel using OpenMP, dividing it into chunks, and counts the number of zero-valued elements.
    - ➢ **Parameters:**
        - `sparseMatrix* mat`: Pointer to the matrix object.
        - `ofstream& outFile`: Output stream to log results.
    - ➢ **Implementation:**
        - Divides the matrix into `K` chunks, with each chunk processed by a separate thread.
        - Counts the number of zeros in each chunk and aggregates the results.
        - Measures the execution time for processing and averages it over multiple iterations.
- ❖ **`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

- ❖ **OpenMP Parallelism:**
    - ➢ **Usage**: The `chunk` function utilizes OpenMP to parallelize the process of counting zeros in different chunks of the matrix.
    - ➢ Key Directive:
        - #pragma omp parallel for num_threads(mat->K) reduction(+:totalSum)
            - Distributes the work of processing chunks across `K` threads.
            - Uses a reduction clause to accumulate the total sum of zero elements across threads.
- ❖ **Thread Count**
    - ➢ Varies between different experiments, with `K` being set to 2, 4, 8, 16, and 32.

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

- **`mixed_omp(sparseMatrix* mat, ofstream& outFile)` Function**
  - **Purpose:**Counts the number of zero-valued elements in the matrix using OpenMP for parallel processing.
  - **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:**
    - Uses OpenMP to parallelize the counting of zero-valued elements across multiple threads.
    - Calculates the total number of zeros and records the count for each thread.
    - Measures the execution time over multiple runs for averaging.
    - Outputs the results to the provided file stream.

2.2.3 Thread Management

- **OpenMP Parallelization**
  - Strategy:
    - Uses OpenMP to parallelize the counting of zero-valued elements in the matrix.
    - Each thread operates on different rows of the matrix to distribute the workload evenly.
- **Thread Count**
  - The number of threads is set by the K parameter in the sparseMatrix object.
  - The thread count is varied from 2 to 32 in powers of 2 during the execution of the processingFiles function.

3 Dynamic technique :

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.
- **`void dynamic_omp(sparseMatrix* mat, ofstream& outFile)` Function:**
  - **Purpose:**
    - This function performs parallel counting of zero-valued elements in the matrix using OpenMP with dynamic scheduling.
  - **Arguments:**
    - `mat`: Pointer to the `sparseMatrix` object.
    - `outFile`: Output file stream to log results.

- **Implementation:**
  - **Parallel Processing:**
    - The loop uses dynamic scheduling with a chunk size defined by `mat->rowInc`, allowing threads to handle uneven workloads more efficiently.
  - **Reduction:**
    - The total sum of zero-valued elements across all threads is accumulated using the `reduction(+:totalSum)` clause.
  - **Thread-specific Counting:**
    - Each thread's count of zero-valued elements is also recorded in a vector (`threadCount`) to track the workload distribution.

3.2.3 Thread Management

- **OpenMP Directives**
  - The program uses the `#pragma omp parallel for` directive to parallelize the loop that counts zero-valued elements in each row of the matrix.
  - **Dynamic Scheduling:**
    - The `schedule(dynamic, mat->rowInc)` clause in the OpenMP directive distributes iterations (rows of the matrix) dynamically among threads with a chunk size of `rowInc`.
    - This approach allows for flexible workload distribution, especially useful when rows have varying numbers of zero elements.
  - **Reduction Clause**
    - The `reduction(+:totalSum)` clause ensures that the total sum of zero-valued elements is safely computed across all threads by combining the results from each thread at the end of the loop.

4 Experiments:

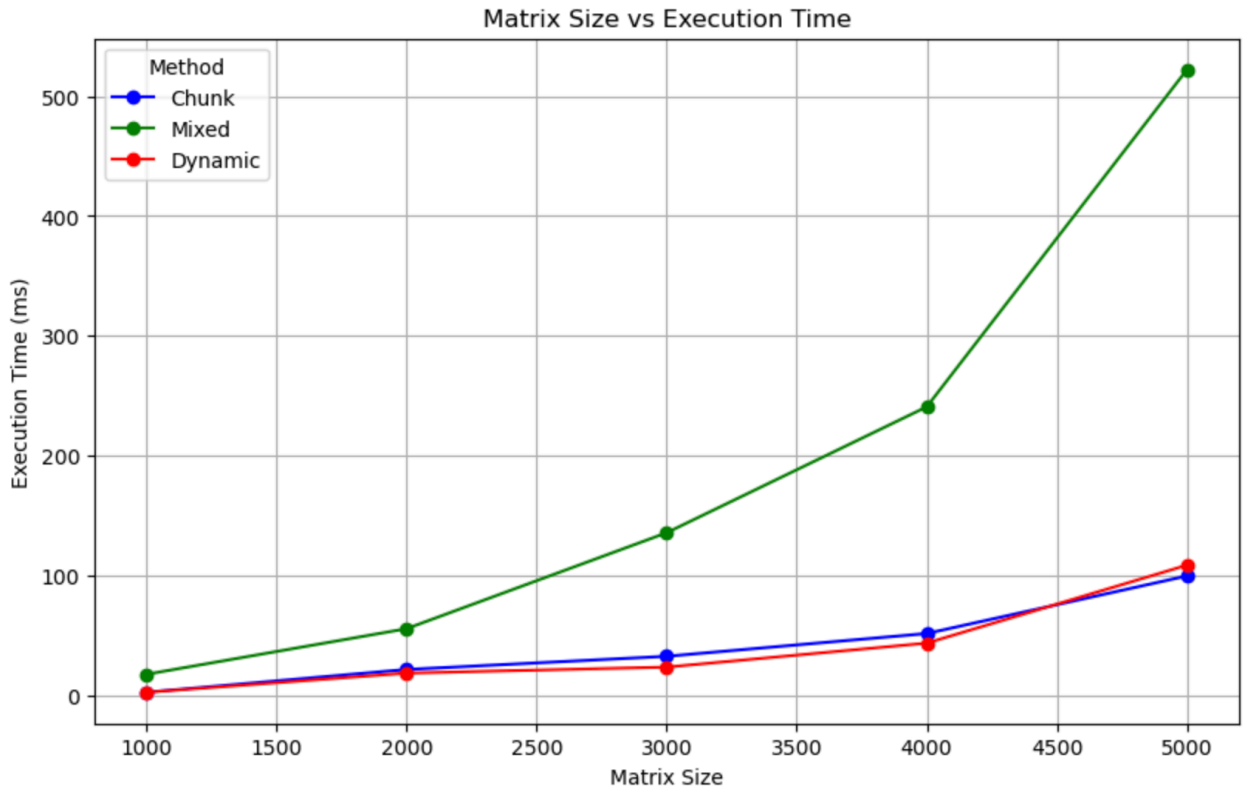
4.1 Time vs Size, N :

- ❖ 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

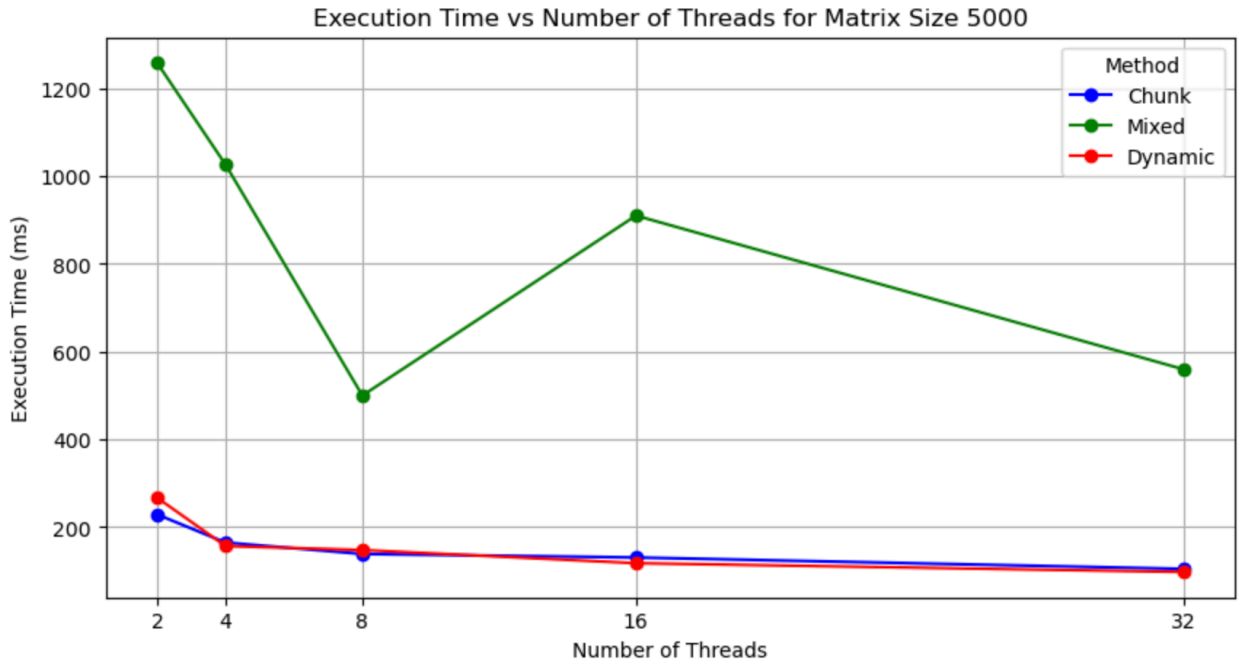


4.1.1 Observation

- **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 It also beats the Dynamic method by a small margin for matrix size 5000 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 grows, 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.
- **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:

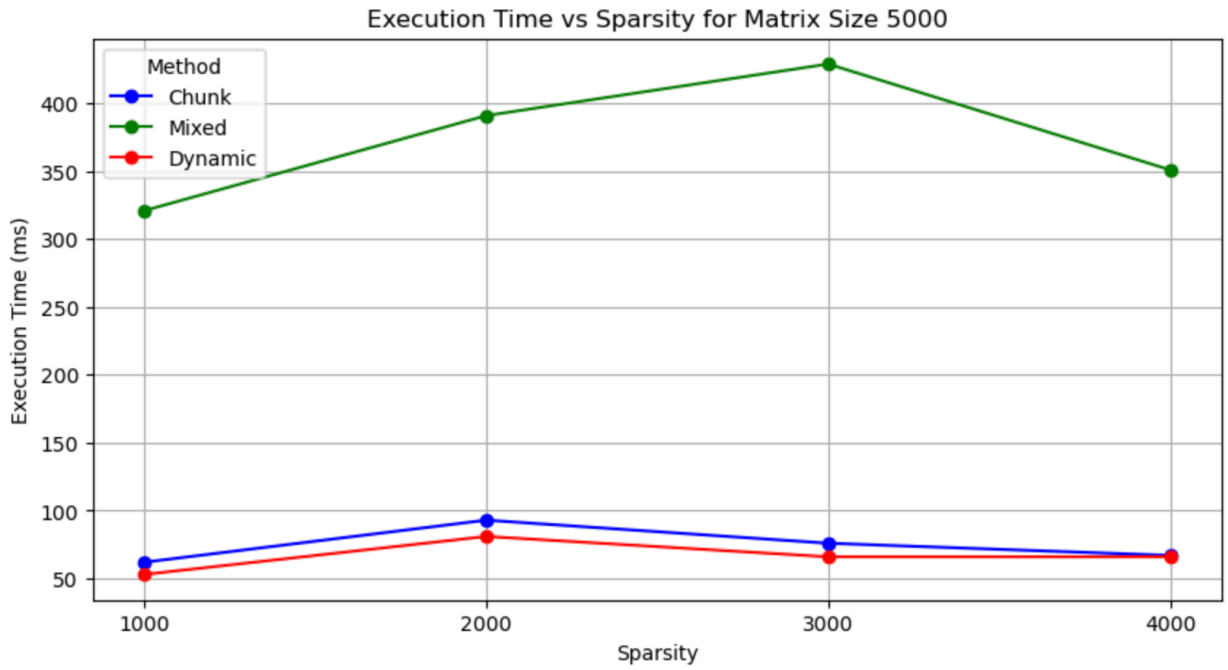
- **Overall Trend:**
  - All methods show a decrease in execution time as the number of threads increases, which is expected as more threads typically lead to better parallelism and faster execution.
- **Chunk Method (Blue Line):**
  - The execution time remains relatively low and stable across all thread counts.
  - The slight decrease in execution time from 2 to 4 threads suggests improved parallel efficiency.
  - After 4 threads, the execution time remains nearly constant, indicating that the Chunk method has effectively balanced the workload among threads. The performance does not significantly benefit from increasing the number of threads beyond a certain point (likely due to overhead or diminishing returns on additional parallelism).
- **Mixed Method (Green Line):**
  - The Mixed method shows a more variable execution time as the number of threads increases.
  - The execution time is high when using 2 threads, suggesting potential inefficiency or poor load balancing at low thread counts.
  - As the number of threads increases from 2 to 8, the execution time drops significantly, indicating better load balancing or parallel efficiency.
  - However, at 16 threads, there is a spike in execution time, suggesting possible contention, overhead, or imbalance at this specific thread count. This spike could be due to increased communication overhead or synchronization issues.
  - Beyond 16 threads, the execution time decreases again, but it remains higher than the Chunk and Dynamic methods, indicating that the Mixed method might not be as effective as the others in this scenario.
- **Dynamic Method (Red Line):**
  - The Dynamic method shows an initial decrease in execution time from 2 to 4 threads, suggesting better parallel efficiency with more threads.
  - The execution time remains relatively constant and low from 4 to 32 threads, similar to the Chunk method.
  - This behavior suggests that the Dynamic method is effective at distributing the workload across varying numbers of threads. The stable execution time also indicates good scalability with an increasing number of 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.
  - 8 threads yield the best performance with the lowest execution time before the performance deteriorates at 16 threads. This makes 8 threads the clear optimal choice for the Mixed method.
- **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



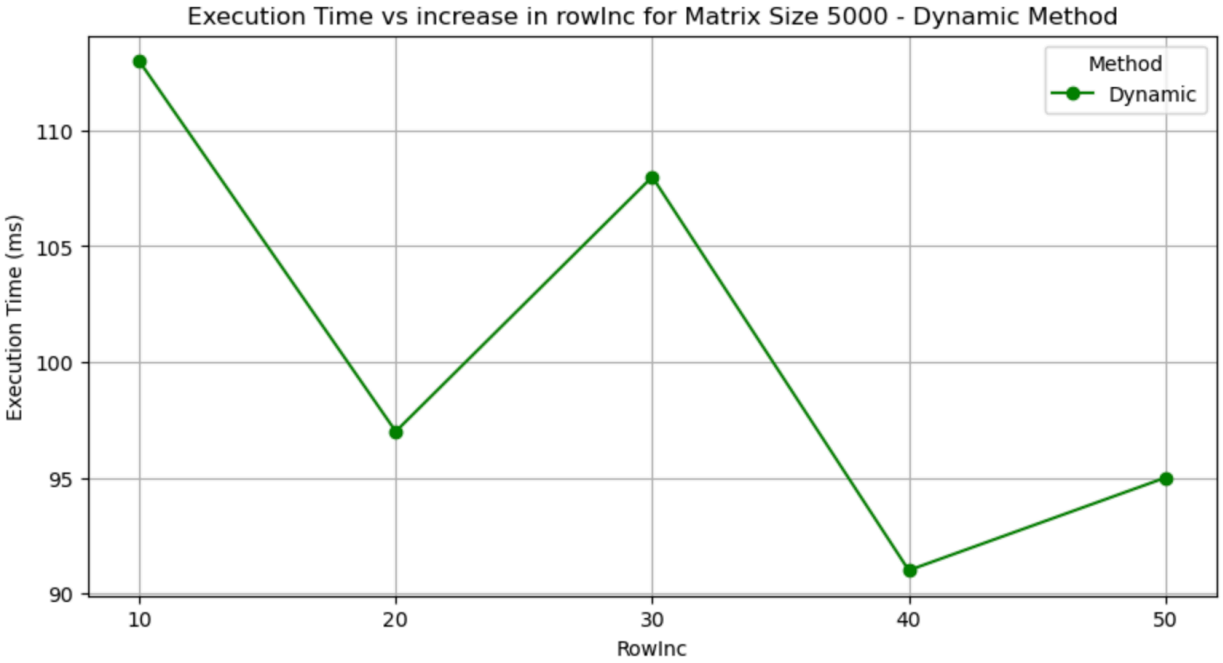
4.3.1 Observations:

- **Execution Time for Chunk and Mixed Methods:**
  - The execution time for the **Chunk** (blue line) and **Mixed** (green line) methods shows an increasing trend initially as sparsity increases, but it slightly decreases as the sparsity reaches higher values.
  - **Chunk Method** maintains a relatively low and consistent execution time across all sparsity levels, ranging from around 50 ms to 100 ms.
  - **Mixed Method** shows a more noticeable increase in execution time, peaking at around 400 ms for certain sparsity levels before decreasing again.
- **Execution Time for the Dynamic Method:**
  - The **Dynamic Method** (red line) shows a similar trend to the Chunk method with a slight increase in execution time as sparsity increases, but it remains relatively low overall.
  - The execution time for the Dynamic method stays close to the Chunk method, ranging from 50 ms to 100 ms across different sparsity levels, indicating efficient handling of varying sparsity levels.
- **Comparison Across Methods:**
  - The Chunk and Dynamic methods perform similarly well, with low execution times that do not vary significantly with changes in sparsity.
  - The Mixed Method, however, shows the highest variability in execution time, with a peak that is significantly higher than the other two methods. This indicates that the Mixed method may not **be as well-suited for handling different sparsity levels efficiently.**
- **Impact of Sparsity on Execution Time:**
  - As the sparsity of the matrix increases (from 20% to 80%), the **Mixed Method** experiences a noticeable increase in execution time, peaking around 60% sparsity, after which it starts to decline.
  - The **Chunk** and **Dynamic** methods show a much more stable execution time across the entire range of sparsity levels, suggesting that these methods are less affected by changes in sparsity.
- **Chunk Method Efficiency:**
  - The **Chunk Method** is highly efficient, maintaining a low execution time regardless of the sparsity level. This suggests that the method is effective at distributing the workload evenly across threads, minimizing any overhead associated with increasing sparsity.
- **Mixed Method Performance:**
  - The **Mixed Method** exhibits a less stable performance, with a significant peak in execution time at around 60% sparsity. This suggests that the method may face challenges when dealing with certain sparsity levels, possibly due to overhead in managing a mix of chunk sizes or dynamic adjustments.
- **Dynamic Method Challenges:**
  - While the **Dynamic Method** generally performs well, maintaining low execution times similar to the Chunk method, it may face some challenges at certain sparsity levels. However, these challenges do not result in significant increases in execution time, indicating robustness in handling varying sparsity levels.
- **Conclusion:**
  - Overall, the **Chunk** and **Dynamic** methods are more robust and efficient across varying sparsity levels, showing minimal changes in execution time.
  - The **Mixed Method** has a more variable performance, suggesting it may not be the best choice when handling a wide range of sparsity levels.

- The choice between **Chunk** and **Dynamic** methods may depend on other factors such as ease of implementation and specific workload characteristics, but both methods are suitable for handling sparsity effectively.

4.4 Time vs. row Increment, rowInc:

- Sparsity of the matrix(S): 40%
- Matrix Size: 5000
- Number of threads(K): 1



4.4.1 Observations:

- **Execution Time at Different `rowInc` Values:**
  - **rowInc = 10**: The execution time is relatively high, at approximately 112 ms. This suggests that smaller increments may result in less efficient workload distribution among threads.
  - **rowInc = 20**: The execution time drops significantly to around 95 ms. This indicates that increasing the increment allows for better workload division, leading to reduced execution time.
  - **rowInc = 30**: The execution time rises again to around 110 ms, suggesting that an increment of 30 may introduce some inefficiencies, possibly due to an uneven distribution of the remaining rows among the threads.
  - **rowInc = 40**: The execution time drops sharply to around 92 ms, the lowest point in the graph. This suggests that this increment provides the most optimal balance of workload among the threads.
  - **rowInc = 50**: The execution time increases slightly to around 98 ms. Although still low, it is slightly higher than the execution time at `rowInc = 40`, indicating that too large an increment can lead to a slight loss in performance.

- **Impact of `rowInc` on Dynamic Method Performance:**
  - The results show that there is an optimal range for `rowInc` (between 20 and 40) where the execution time is minimized. This suggests that the efficiency of the Dynamic Method is highly sensitive to the value of `rowInc`.
  - Smaller `rowInc` values (like 10) may lead to more frequent context switching or overhead in managing smaller chunks, thereby increasing execution time.
  - Larger `rowInc` values (like 50) can lead to fewer chunks and possibly better cache utilization, but beyond a certain point, the efficiency may decrease due to uneven workload distribution.
- **Conclusion:**
  - The optimal `rowInc` for minimizing execution time in this experiment appears to be around 40, where the Dynamic Method shows the best performance.
  - Both too small and too large increments can lead to suboptimal performance due to the trade-offs in workload distribution and management overhead.
  - Fine-tuning the `rowInc` parameter is crucial for achieving optimal performance with the Dynamic Method when processing matrices with varying levels of sparsity.

5 Conclusion:

In this assignment, we evaluated three methods—Chunk, Mixed, and Dynamic—for counting zero-valued elements in a square matrix using OpenMP, comparing their performance across varying matrix sizes, thread counts, sparsity levels, and row increments. The Chunk and Dynamic methods consistently demonstrated efficient performance with stable and low execution times, making them well-suited for parallel processing across different matrix configurations. In contrast, the Mixed method showed greater variability and inefficiency, particularly with larger matrix sizes and higher thread counts. Overall, the Chunk and Dynamic methods are preferable for robust and scalable parallel computation, while the Mixed method may require further optimization for consistent performance.