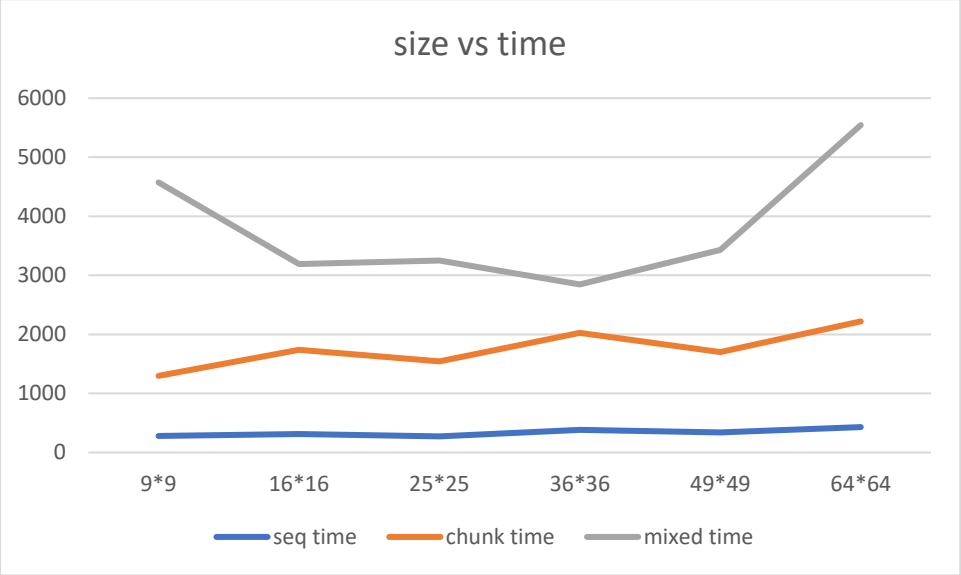
REPORT – SUDOKU MULTITHREADING

Experiment 1:
In this experiment, you have to keep the number of threads constant say 8 and compare the time taken to validate the sudoku by varying the size as follows: 9X9,16X16, 25X25, 36X36, 49X49 and 64X64. Thus, in this experiment, the size of the sudoku will be on the x-axis(in microseconds) as described above and the y-axis will show the time taken.

Graph:



Explanation:

Observed Trends:

1. **Sequential Time (Blue Line):**

- The sequential time is the lowest for all grid sizes compared to chunked and mixed approaches.
- This indicates that for the given problem size, the sequential approach is more efficient because it avoids the overhead associated with multithreading.

2. **Chunked Time (Orange Line):**

- The chunk time is consistently higher than the sequential time. It gradually increases as the grid size increases, reflecting the impact of thread management overhead.

- Even though chunk is designed to distribute work across multiple threads, the problem size isn't large enough to offset the cost of managing threads, leading to slower execution.

3. **Mixed Time (Gray Line):**

- The mixed time is the highest and fluctuates significantly as grid size increases.
- This could be due to inefficient workload balancing or additional overhead from combining strategies (e.g., managing both chunking and sequential tasks within threads).

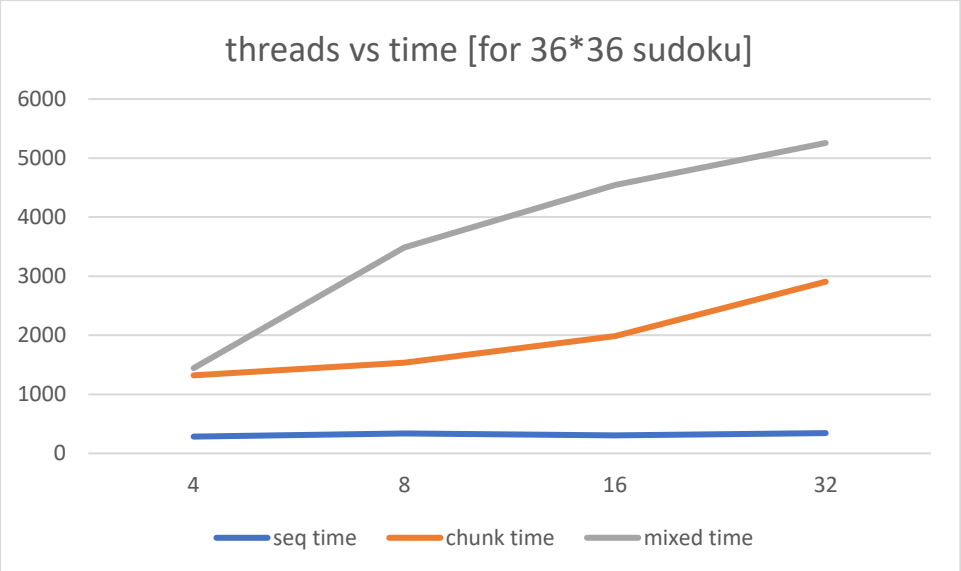
Why multithreading is slower here?

Both chunk and mixed approaches introduce thread management, synchronization, and communication overheads. For relatively small problem sizes (like those in this experiment), these overheads outweigh the benefits of parallelism.

Multithreading typically benefits large problem sizes, where the workload per thread is substantial enough to justify the overhead. Here, the Sudoku grid sizes (e.g., 9x9 to 64x64) don't provide enough workload for threads to operate efficiently.

Experiment 2:
You have to keep the size of the sudoku constant 36X36 and compare the performance by varying the number of threads from 4 to 32 in multiples of 2: 4, 8, 16 and 32. In this experiment, the number of threads will be on the x-axis and the y-axis will show the time taken.

Graph:



Explanation:

Sequential Time (Blue Line):

- **Observation:** The sequential approach maintains almost a constant time regardless of the number of threads because it does not rely on multithreading.
- The sequential method avoids multithreading overhead such as thread creation, synchronization, and task division. It processes the grid linearly, which is inherently efficient for a task size like a 36x36 Sudoku.

Chunk Time (Orange Line):

- **Observation:** The chunk approach shows an increase in time as the number of threads increases, but it is consistently better than the mixed approach.
- The chunk method divides the grid into independent chunks and assigns them to threads. As the number of threads increases, thread management overhead grows, but the task is still efficiently distributed. The chunk strategy is simpler and has less synchronization overhead compared to the mixed approach, leading to better performance.

Mixed Time (Gray Line):

- **Observation:** The mixed approach has the highest time across all thread counts and increases significantly as the number of threads grows.
- The mixed approach leads to higher synchronization and communication overhead. This inefficiency becomes worse as the number of threads increases, causing poor scaling and higher overall time.

The chunk approach minimizes synchronization overhead by dividing the work into larger, independent tasks that threads can process without much interaction.

The mixed approach introduces complexity in managing workload, often causing threads to idle while waiting for others to complete their tasks. This poor load balancing results in higher execution time.

The sequential method avoids all forms of multithreading overhead, such as thread creation, context switching, and synchronization.

For a grid size like 36x36, the workload is not large enough to justify the cost of multithreading, making the sequential approach the most efficient.

MY APPROACH – CODE EXPLANATION:

Chunk :

Chunk method effectively divides the work into smaller tasks (chunks), where each thread handles specific chunks of rows, columns, and subgrids.

Initially reading the value of K, N and the sudoku to be validated from the input file – 'inp.txt' , K is then divided into K1, K2, K3 for validating rows, columns and subgrids according to the chunk algorithm.

Chunk size is given by R/K1 for Row validation, C/K2 for column validation and n/K3 for subgrid validation.

log_buffer holds logs for each thread, storing the message and timestamp.

is_valid is a flag to track the validity of the Sudoku grid.

Each thread logs its operation (row, column, or subgrid validation) with a timestamp.

The logs are stored in log_buffer and sorted based on the timestamp.

Validation Functions:

- **validate_row()**: Validates a given row for duplicate values.
- **validate_column()**: Validates a given column for duplicate values.
- **validate_subgrid()**: Validates a subgrid (square block) of the Sudoku grid for duplicates.

Thread Functions:

- **row_runner()**: Each thread validates a chunk of rows (as per the chunk method).
- **col_runner()**: Each thread validates a chunk of columns.
- **subgrid_runner()**: Each thread validates a chunk of subgrids.

Main program flow:

The program reads the number of threads (K) and Sudoku size (N) from the input file.

The threads are created, and the work is divided between them based on the chunk sizes (R/K1 for rows, C/K2 for columns, and n/K3 for subgrids).

After validation, the threads are joined, logs are sorted, and results are printed in an output file, including the validation result and the execution time.

Mixed:

The Mixed method distributes the validation workload among threads in a purely cyclic manner. This ensures that the threads take turns validating non-consecutive rows, columns, and subgrids, leading to a balanced workload.

The program reads the values of KKK (number of threads), NNN (Sudoku size), and the Sudoku grid from the input file (inp.txt).

The threads are assigned to validate rows, columns, and subgrids in a cyclic fashion:

- For rows: Thread 1 validates rows 1, K1+1, 2K1+1 and so on. Thread 2 validates rows 2, K1+2, 2K1+2 , and so forth.
- For columns: Thread 1 validates columns 1, K2+1, 2K2+1, and so on. Similarly, Thread 2 validates 2, K2+2, 2K2+2 , and so on.
- For subgrids: Subgrids are validated cyclically by threads in the same way as rows and columns.

The functions and the key variables like the log_buffer and flags are similar to the Chunk method .

Main function flow:

Read the values N, K, and the Sudoku grid from the input file (inp.txt).

Divide K into K1, K2, and K3 for rows, columns, and subgrids.

Assign row, column, and subgrid validation tasks to threads cyclically. Start threads for rows, columns, and subgrids.

Threads log their operations and outcomes with timestamps in log_buffer. Threads join after completing their tasks. Logs are sorted by timestamp for organized analysis. The program writes the validation results (valid or invalid) and execution time to an output file (outputmixed.txt).

Sequential:

No threads are used in this code. The entire program runs sequentially as a single thread, performing validation for rows, columns, and subgrids one after another. No parallel execution or pthreads is implemented here.
It runs exactly like the other two methods, the input and output functionality, use of clock to calculate the execution time , validation rows, columns and subgrids works the same but processes the workload sequentially rather than distributing it among multiple threads.

As a result, all validations (rows, columns, and subgrids) are completed one at a time, which can lead to longer execution times for larger Sudoku grids due to the absence of parallelism though it avoids the complexities of thread management.