Operating Systems - II (CS3523)

Jan 29, 2025

Programming Assignment 1: Validating Sudoku Solution

Chunk Algorithm:

The multithreaded Sudoku validator using the chunk method was implemented by validating the rows, columns, and subgrids. In this code, I used **pthread** (POSIX threads) to divide the task into multiple threads for better performance and efficiency, particularly for larger Sudoku grids.

The validation is done by checking rows, columns, and subgrids.

Code Structure:

1. `struct thread_data()`:

   This structure holds the parameters such as rows, columns, and subgrids, which are used for checking the validity of Sudoku. It also stores the output messages produced by the thread.

2. `row_check()` **function**:

   Each thread checks a subset of rows using `start_row` and `end_row` parameters and checks if the values are unique using the `bool` array `visited`.

3. **`col_check()` function**:

   Similar to the `row_check()` function, this function does the same task but for the parameters `start_col` and `end_col`, which define the starting and ending columns for the thread to check a subset of columns.

4. **`subgrid_check()` function**:

   Each thread checks a subset of subgrids (n x n blocks). For each subgrid, the thread ensures that all values in the subgrid are unique.

5. **Main Function**:

   The main function reads the Sudoku grid from a file and initializes the threads. It divides the grid into chunks and creates threads for row, column, and subgrid validation. After all threads finish, the program checks the overall validity of the grid and writes the results to an output file.

Overview of Process:

1. The program reads the dimensions of the Sudoku grid (K, N) from an input file (`input.txt`).
2. Threads are created for row, column, and subgrid checks using the `pthread_create()` function.
3. Each thread executes either the `row_check`, `col_check`, or `subgrid_check` function.

4. The program uses `pthread_join()` to wait for all threads to finish before proceeding.
5. The output file contains the validity of each row, column, and subgrid, as well as the overall validity of the Sudoku puzzle and the time taken for validation.
6. The time taken to perform the validation is measured using the **chrono** library and is written to the output file in microseconds.

Mixed Algorithm:

The program divides the task into smaller parts by using multiple threads, thus improving the overall performance by parallelizing the validation process.

Code Structure:

1. `struct thread_data()`:

   This structure holds the parameters for each thread, including the Sudoku grid, thread ID, and output vectors for row, column, and subgrid results. Each thread stores the results of the row, column, or subgrid it checks in these vectors.
2. `row_check()` **function**:

   Each thread processes a subset of rows. It checks for the uniqueness of elements in the row; if not unique, the row is marked as invalid.

3. `col_check()` **function**:

   Similar to row validation, but each thread processes a subset of columns and checks for uniqueness in each column.

4. `subgrid_check()` **function**:

   Each thread processes a subset of subgrids (n x n blocks). It checks each subgrid to ensure all values are unique within the block.

5. **Main Function**:

   Similar to the Chunk method, the main function reads the Sudoku grid from the input file and initializes threads for row, column, and subgrid validation.

Observations from the graphs:

EXPERIMENT 1-

For **Experiment 1**, the goal is to analyze the performance of the Sudoku validator by keeping the number of threads constant (8 threads) and varying the size of the Sudoku grid. The grid sizes to be used are: 9x9, 16x16, 25x25, 36x36, 49x49, and 64x64. The x-axis will represent the size of the Sudoku grid, and the y-axis will represent the time taken (in microseconds) for the validation process.

From the graph below, which was taken from the code, we can observe that the **mixed method** has good performance and efficiency, making it the best among the three

processes. As we can see, the time taken by the mixed method increases very slightly compared to the other two methods, which is evident from the graph.

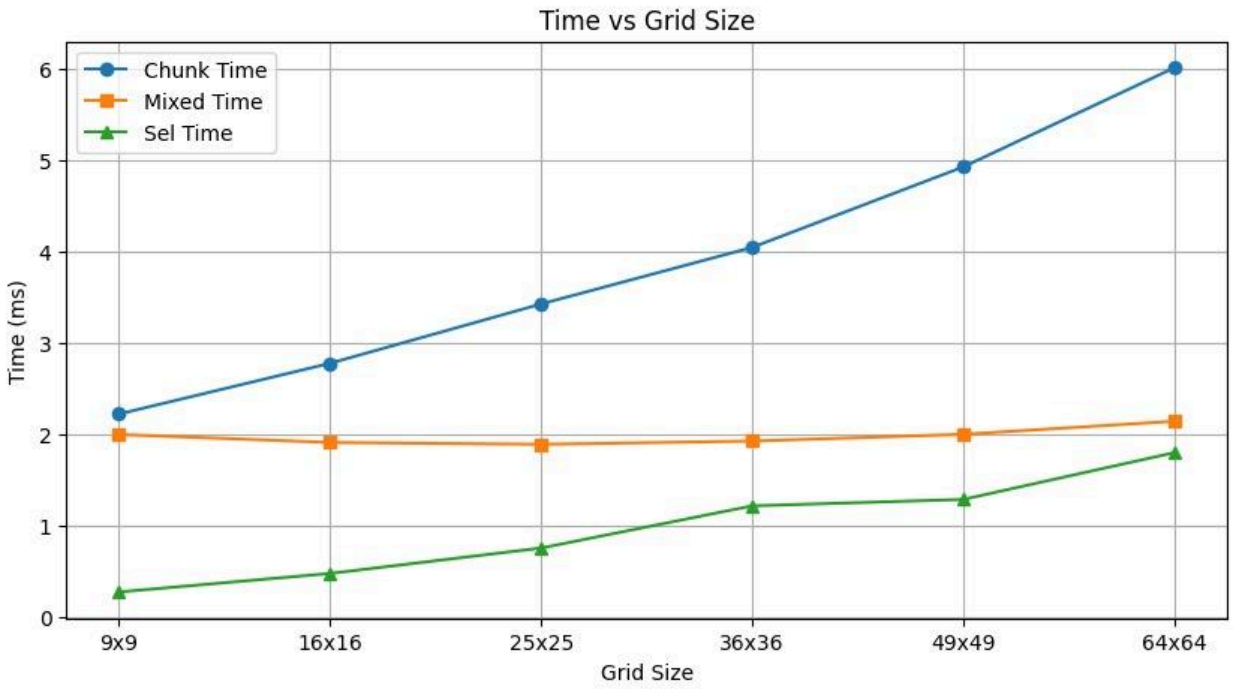
On the other hand, the **chunk method** shows that its time increases almost linearly, making it unsuitable for Sudoku puzzles with larger grid sizes.

When it comes to **sequential time**, it performs very well for small grid sizes. However, as seen in the graph, the time increases almost linearly after reaching a 25x25 grid size.

Conclusion:

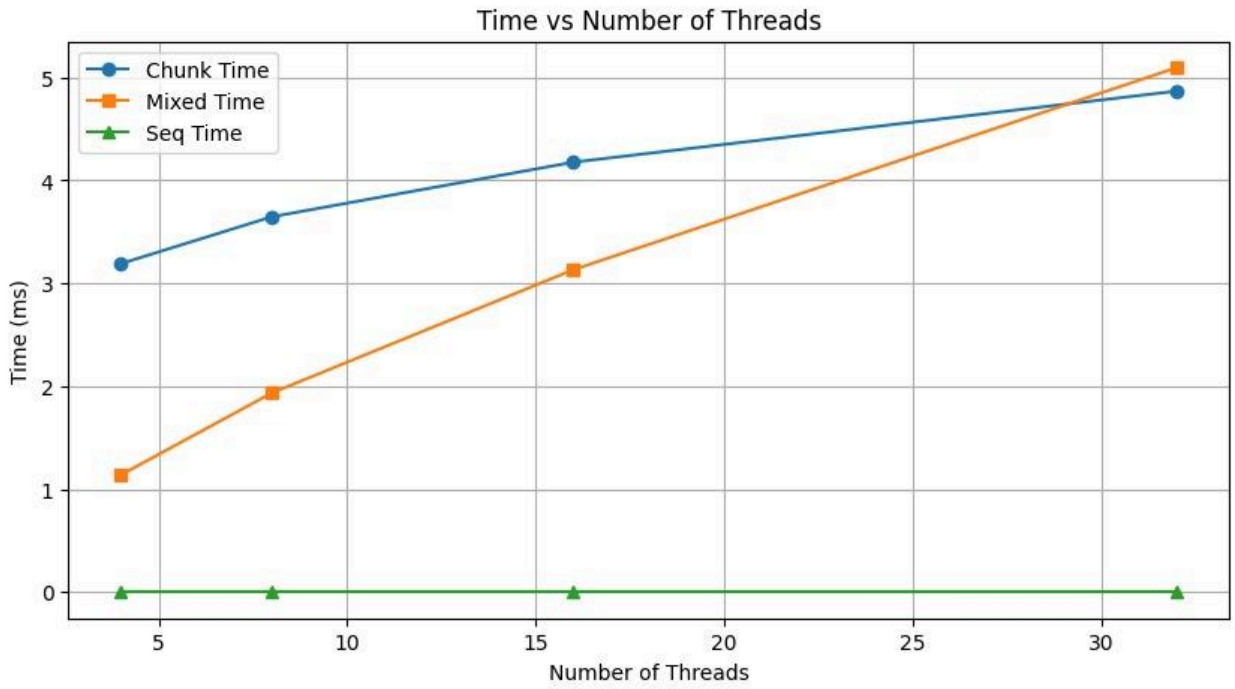
For better performance:

- It is recommended to use **sequential** for small grid size Sudoku puzzles.
- The **mixed algorithm** should be used for larger grid sizes. If a hybrid approach is possible, that would be ideal; if not, the **mixed algorithm** is the better choi



EXPERIMENT 2–

As we can observe from the graph below, the sequential method remains unaffected by the number of threads, as it uses only one thread. For the chunk process, the situation worsens as the number of threads increases, but it still performs better than the mixed method when a high number of threads is considered. Therefore, using the mixed method for a few threads and the chunk method for higher thread counts is more efficient. A general observation is that performance improves when using a suitable number of threads. Both too few or too many threads result in inefficient performance for larger Sudoku grids. The optimal number of threads varies with grid size.


Time vs Number of Threads
Chunk Time
Mixed Time
Seq Time
Time (ms)
Number of Threads
0
1
2
3
4
5
5
10
15
20
25
30