Programming Assignment 1: Validating Sudoku Solution

Uppu Eshwar
Roll No.: Ch21btech11034

1 Design of the Problem

The values of N (size of Sudoku), K (number of threads), and the values of Sudoku are taken from `input.txt` (input file). The clock is started to calculate the time taken to complete the process. K threads are created using a loop and assigned the functions "Cyclic Runner" and "Chunk Runner," respectively.

There are six important functions:

- `rowCheck`: Checks the validity of a row in Sudoku.
- `colCheck`: Checks the validity of a column in Sudoku.
- `subCheck`: Checks the validity of a subgrid in Sudoku.
- `CyclicRunner`: Function that threads work on in cyclic distribution.
- `ChunkRunner`: Function that threads work on in chunk distribution.
- `main`: Main function.

Each function uses an array of size N, initialized to zero, to check for all N numbers. If a number is present, its corresponding index is marked. If the number is greater than N, less than 1, or already present, the function returns -1. Otherwise, it returns the respective row, column, or subgrid number.

A class `thread_data` is created to store the start column, row, subgrid number, and the number of elements each thread checks. It also contains a `terminate` flag for early exits.

A buffer vector is implemented so that each thread maintains its own buffer. All log statements, including timestamps, are stored in the buffer to ensure proper ordering when written to a file.

1.1 Working of Chunk Runner

Threads precompute their working ranges in local storage. The chunk runner iterates through its assigned region, validating rows, columns, and subgrids. If discrepancies are found, the thread exits via `pthread_exit(0)`.

1.2 Working of Cyclic Runner

Similar to the chunk runner, cyclic runner precomputes its range. It iterates over allocated regions and exits upon detecting inconsistencies.

1.3 Source Code Methods

The source code contains all three methods (sequential, chunk, cyclic) in a single file.

2 Experimental Results

2.1 Experiment 1: Sudoku Size vs. Execution Time

Size ($N \times N$)	Chunk (μs)	Cyclic (μs)
9	822.80	941.40
16	885.60	865.20
25	954.20	899.20
36	958.60	937.20
49	1009.40	977.40
64	1130.00	1088.20
81	1235.20	1249.40

Table 1: Execution Time vs Sudoku Size

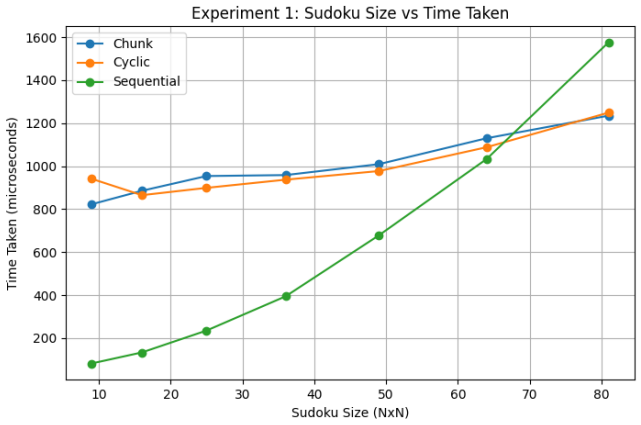

Figure 1: Execution Time vs Sudoku Size

Observations:

- Sequential execution was significantly faster for small grid sizes due to lower overhead.
- As grid size increased, sequential execution became exponentially slower, whereas multithreading scaled efficiently.
- Chunk-based and cyclic methods had comparable performance, with chunk-based being slightly faster in most cases.

2.2 Experiment 2: Number of Threads vs. Execution Time

Threads	Chunk (μs)	Cyclic (μs)
4	1316.40	1376.20
8	1338.80	1412.60
16	1782.80	1723.80
32	2793.40	2636.00
64	5240.60	5155.60

Table 2: Execution Time vs Number of Threads

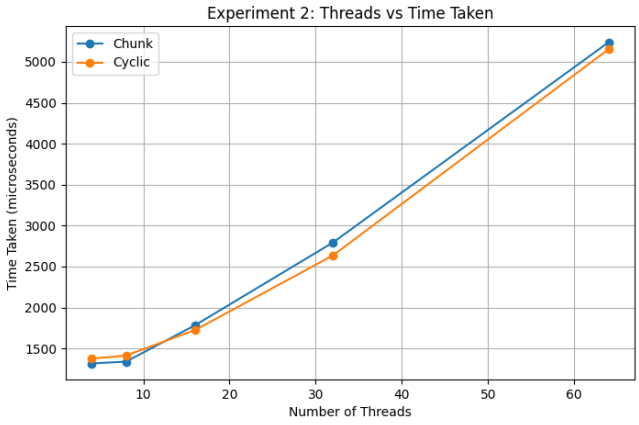

Figure 2: Execution Time vs Number of Threads

Observations:

- Execution time increased as the number of threads increased due to overhead.
- Cyclic scheduling performed marginally better where load balancing played a role.

3 Analysis of Results

- **Why was sequential execution faster for smaller grids?**
  - Overhead from thread creation outweighed benefits of parallelism.
  - Sequential execution benefited from CPU cache locality and minimal context switching.
- **Why did multithreading not scale well beyond 32 threads?**
  - Excessive context-switching and thread contention degraded performance.
  - The problem size was insufficient to fully utilize high thread counts.