# Homework 2

Ege Aktemur

May 2023

## 1. Implementation (Algorithm Overview and Heuristics)

Implementation includes a backtracking algorithm that attempts all possible numbers (1 to 9) in a cell and moves on to the next cell if the current partial solution is still valid. If the partial solution becomes invalid, it backtracks to the previous cell and tries the next partially valid number. This process continues until a solution is found or all possibilities are exhausted.

The verifyRemainingNumbers() function checks whether the remaining numbers needed for the sum can be achieved using the remaining empty cells, considering the constraints of the puzzle. It does so by calculating the minimum and maximum possible sums with the given constraints.

The wrongSolution() function checks whether the current partial solution is incorrect, considering the current sum, the direction of the sum, and the cells involved.

The solve_kakuro() function is the main backtracking algorithm implementation. It recursively attempts all possible numbers in a cell and moves to the next cell if the partial solution is still valid.

The solution() function initializes the data structures, assigns cell values, and calls the solve_kakuro() function to start the solving process.

## 2. Execution times

Small board sizes: As the number of threads increases, the execution time tends to increase as well, except for some cases. This is likely due to the overhead of creating and managing threads.

Medium board sizes: The execution times for varying numbers of threads do not follow a clear pattern, with some cases improving as the thread count increases, while others show the opposite trend. This suggests that the optimal number of threads may depend on the specific puzzle.

Large board sizes: In general, the execution times tend to decrease as the number of threads increases, with a few exceptions. This suggests that parallelism becomes more beneficial as the problem size grows.

It's worth noting that the optimal number of threads is not consistent across different problem sizes and specific puzzles which might be due to various factors such as thread management overhead or specific characteristics of the puzzle.

| | Serial | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|---|
| 3_1 | 0.000007 | 0.000056 | 0.000082 | 0.000097 | 0.000114 | 0.000206 |
| 3_2 | 0.000013 | 0.000067 | 0.000068 | 0.000093 | 0.000121 | 0.000276 |
| 3_3 | 0.000009 | 0.000051 | 0.000054 | 0.000075 | 0.000172 | 0.000194 |
| 4_1 | 0.000024 | 0.000146 | 0.000135 | 0.000182 | 0.000218 | 0.000249 |
| 4_2 | 0.000093 | 0.000411 | 0.000437 | 0.000571 | 0.000703 | 0.000775 |
| 5_1 | 0.000016 | 0.000101 | 0.000087 | 0.000281 | 0.000153 | 0.000226 |
| 5_2 | 0.000045 | 0.000254 | 0.000207 | 0.000231 | 0.000277 | 0.000323 |
| 20_1 | 0.005781 | 0.069562 | 0.056865 | 0.076777 | 0.100631 | 0.129345 |
| 30_1 | 0.352886 | 6.221817 | 8.469823 | 7.770874 | 9.511361 | 10.473709 |
| 40_1 | 9.246898 | 370.926425 | 373.464601 | 325.998019 | 337.051251 | 446.160361 |

# 3. Efficiency Tricks

## a. Spatial and temporal locality

### Bitset for spatial locality

A std::bitset is used to store the "exists" variable in the verifyRemainingNumbers and wrongSolution functions. This data structure provides spatial locality by packing boolean values into a contiguous block of memory, allowing for faster access times and cache performance.

### DeepCopy for temporal locality

The deepCopy function is used to create a copy of the sol_mat matrix before modifying it. This ensures that the original matrix remains unchanged, allowing for a more efficient backtracking algorithm.

### Exploit spatial locality in loops

Loops in the verifyRemainingNumbers and wrongSolution functions have been designed in such a way that they access elements in a contiguous memory region. This helps to improve cache performance and overall efficiency.

## b. Preprocessing

Preprocessing which sums belong to which cell: The which_sums_cell_exists data structure is a 3D vector that maps each cell to the sum objects it belongs to. This preprocessing step helps to speed up the solving process by allowing the solver to quickly identify which sums are associated with a given cell.

## c. Search Space Optimizations

### Pruning

In the solve_kakuro function, the search space is pruned by checking whether the current cell's value is valid (i.e., it doesn't cause any conflict with the given sums). This helps to reduce the total number of possibilities that need to be explored during the solving process.

### Backtracking

The backtracking algorithm is used to explore the solution space. By incrementally building the solution and undoing any changes when a conflict is found, the algorithm can search for a solution more efficiently than a brute-force approach *

## d. Parallelism

The code uses the OpenMP to parallelize the computation of valid numbers in solve_kakuro function. To elaborate on that more while in the serial algorithm code it calculates the next number if the current one is not valid, in the parallel algorithm it calculates all of the numbers' validness at the start in a parallel manner. While this precalculation seems efficient in large boards high thread numbers compared to 1 thread execution, it is still more inefficient than the serial algorithm since it creates overhead does unnecessary calculations.

## d. Execution Instruction

Code is compiled using: " g++ kakuro_solver_hw2.cpp -O3 -fopenmp"
It can be run by just "./a.out" but one my want to edit the main for testing purposes.