CS 406 Parallel Processing and Algorithms
Homework 2

Rebah Özkoç - 29207

May 9, 2023

1 Introduction

Kakuro is a logic puzzle game like crosswords, but with numbers instead of letters. The objective is to fill a grid with digits from 1 to 9 in such a way that the sum of the digits in each horizontal or vertical sequence matches a given hint number. No digit can be repeated within the same sum. A puzzle can have more than one solution.

In this Kakuro Solver, I implemented a recursive backtracking algorithm to find the solution for the given puzzle. The backtracking algorithm is a depth-first search technique that solves problems by recursively exploring possible solutions, building candidates incrementally, and removing a candidate if it doesn't lead to a valid solution. It "backtracks" when it reaches a dead end and continues searching for a valid solution with the next candidate.

To solve the problem, I first implemented the solution sequentially. After I saw that the sequential implementation works, I tried to improve the performance by parallelizing the tasks and loops.

2 Sequential Solution:

In the implementation, I implemented various auxiliary methods and functions to support the solution function. First, I added some methods to the Sum struct that helps to check the correctness of sums and a method to update the sum array elements.

Some of the methods are:

checkSumPartial() ⇒ If the sum of "arr" is less than or equal to the hint returns true else false

checkSumComplete() ⇒ If the sum of "arr" is equal to the hint returns true else false

areElementsUniqueExceptEmpties() ⇒ checks the uniqueness of "arr" elements except for zeros.

areElementsUnique() ⇒ checks the uniqueness of "arr" elements.

isArrFull()

isLastCell(COORD curr) ⇒ returns True if the currently checked cell is the last element of the sum in the horizontal or vertical direction.

fullCheck(COORD curr) ⇒ Combines the previous methods and checks if the sum obeys all of the puzzle constraints and returns true if it satisfies the conditions.

updateArr(int i, int j, int val) ⇒ adds the value val at the sum if it contains the location matrix[i][j].

In addition, I created another struct namely "mat_iter" to iterate in the matrix to find the next empty (contains -2) cell. This struct initializes at the first empty cell of a given matrix and each call of "set_next()" method updates the current coordinate of the iterator. After the last cell it sets the coordinates to a negative value.

I implemented the solution function with a recursive backtracking algorithm. First, I removed the int** mat from the function signature and added a mat_iter iterator. Then, I added two cases which are the base case and the recursive case. If the iterator passes the last empty cell this means that the algorithm is ended successfully, and this is the base case. If the iterator is on an empty cell it tries to put values from 1 to 9 to the cell. If the value breaks a constraint in any of the sums, this means that the value cannot be on that cell. In this case, the for loop continues and it tries to put a new value at that cell. If the value does not break any constraint in the sums, this means that the value is a candidate for the correct value. In this case, the partial assignment is done for that cell with that value and the iterator continues with the next empty cell recursively. If the solution gets stuck on the next cell, the iterator comes one step back and continues with a new candidate solution from the range 1 to 9. If there is no possible value for a cell from the range from 1 to 9 then the kakuro is not solvable. In this case, the function returns false.

Lastly, I fixed some of the bugs in the given draft code such as the bug in the sum constructor which does not properly initialize the sum array and the loop of the matrix deletion.

2.1 Sequential Solution Timings:

board3_1: 0.000531 seconds

board3_2: 0.000920 seconds

board4_1: 0.002446 seconds

board4_2: 0.049014 seconds

board5_1: 0.003798 seconds

board5_2: 0.029305 seconds

board20_1: 4317.991876 seconds

board20_40 and others: Too long to run. After 3 hours, I stopped the tasks.

3 Parallel Execution

My first optimization was dividing the recursive calls into OpenMP tasks. I called the solution function inside a omp parallel and omp task block in the solution function. The result of the function is stored in a shared boolean result value. This optimization led to an increase in performance by parallelizing the evaluation of different boards at the same time. Also, it did not affect the correctness of the solution boards.

board3_1: 0.001269 seconds

...

board20_1: 1801.406485 seconds

It gave 2.39 times faster on **board20_1**. But on the smaller boards, the parallelization overhead was greater than the improvements.

After that, I tried to parallelize for loops of the sum check methods with parallel sum reduction clauses. These parallelization efforts worsened the execution time because the array lengths are between 4 and 60. And this parallelization causes too much granularity because the loops already belong to different tasks which work in parallel. The execution time for **board20_1** was 3789.295089 seconds. With this result, I decided to remove the reduction clauses.

Then, I improved my code by changing the "full_check()" method. I added another constraint for the candidate values to decrease the recursive calls. While creating the sum structures I calculated the possible minimum value and possible maximum for a cell in that sum with the hint value and length of the sum array. For example, if the sum is 22 and the length of the sum array is 3 then the minimum

value for a cell of this array is 5 because the other elements can be 8 + 9 = 17 in maximum. So, 22 – 17 = 5 gives us the minimum value for a cell. With the same logic, we can calculate the possible maximum value for a cell.

This trick improved my code performance significantly. I got 6.085165 seconds for **board 20_1** with default optimization and I got 1.076514 seconds with -O3 optimization flag.

3.1 Parallel Execution Timings:

Here are the last execution times with -03 flag:

board3_1: 0.000210 seconds

board3_2: 0.000275 seconds

board4_1: 0.000568 seconds

board4_2: 0.002101 seconds

board5_1: 0.000603 seconds

board5_2: 0.001548 seconds

board20_1: 1.076514 seconds

board20_40: 546.348427 seconds

board30_1: 1370.264503 seconds

board30_60 and others: 3+ hours

4 How To Compile

Download the **kakuro_solver_hw2.cpp** file. Then compile with g++ and run it.

```
g++ kakuro_solver_hw2.cpp -O3 -fopenmp

\a.out boardname1.kakuro
```