CS406 HW2

Baturalp Kabadayı

May 2023

1 Introduction

The goal of this homework was to design an implement a parallel algorithm in order to solve kakuro puzzles, using C++ and OpenMP along with it. You can check the GitHub repository for this homework to see all source code and extra samples of kakuro boards: kakuro_solver_omp.

2 Algorithm Description

One approach to solve this problem would be a brute force search to try all possible boards. However, due to the nature of the problem, most of the possibilities can be eliminated if we use a correct search strategy.

First of all, a method is needed to identify which cell is included in which sum. For this purpose, a 3D array is created so as to map board cells to the sums they are included in. This way, we only match cells and sums once and there is no need to find them again and again.

Having that, we start from the first cell, insert digit (d), if the sum is valid, recursively go to next cell with a copy of the board that d was inserted in.

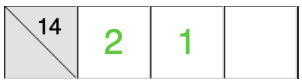
2.1 Checking Sums

A sum being valid means that it is either completed (there are no duplicate elements, and sum of elements is correct) or it has potential to be completed, .i.e not all elements are placed and no error is occurred yet. 3 types of errors are defined, which can disrupt a sum being valid. First is **duplication**, i.e. a digit is included more than once in that sum.

The second type of error is called **over-sum**. This means that the sum of elements placed in a sum, until now, is so large that the sum cannot be valid anymore. Here is an example to explain it better:

The remaining sum for last 2 cells is 1. Even though 1 and 2 (which are the minimum possible) are placed, their sum is 3, which makes this sum being valid impossible.

The last type of error is called **under-sum**. This means that the sum of elements placed in a sum, until now, is so small that the sum cannot be valid anymore. Here is an example to explain it better:



The remaining sum for last cell is 11. Even though 9 (which is the maximum possible) is placed, the sum cannot add up to total 14.

2.2 Main Algorithm

The motivation behind the algorithm is to divide the problem into two tasks in each recursion. It starts with 2 tasks, lets call them task A and task B. Task A is to do the following job with inserting digits (6 to 9) to the current (first) cell. For each digit it looks for the sums that the current cell it is included in and checks the sum with the way described in section 2.1. If the sum is valid, it creates 2 new tasks of type A and B, with passing a copy of the current board and moving to the next cell. Task B is to do the same thing for digits in range (5, 1).

For both tasks A and B, if the sum check returns error **duplicate**, no child tasks are created. Since there is already a duplication, there cannot be any solutions starting with the current board state. They simply move on with the next digit.

However, for task A, if the error is **over-sum**, no child tasks are created and the task itself is halted too. In other words, if task A gets over-sum error for the digit 7, it does not anymore execute the sum checking process for digits 8 and 9. The reason is that the numbers greater than 7 are also guaranteed to get over-sum error.

Similar approach is present for task B. However, since task B executes digits in a decreasing order, if **under-sum** is encountered, no child tasks are created and the task itself is halted too. For instance, if there is an under-sum error for digit 4, sum checking process is not executed for digits 3, 2, 1. The reason again is that the numbers smaller than 4 are also guaranteed to get under-sum error.

With this method, the algorithm can eliminate so many possibilities. Finally, if a task is at the last cell and sum checking process turns out to be successful,

the solution is found. That arrangement of the board is returned.

2.3 Parallelization

Because of the fact that described tasks are independent from each other, they can be executed concurrently. They use different copies of boards so that they do not also have to write on the same memory item. There is only a single limitation which is a task must wait for its child tasks before creating more tasks or returning. This is achieved by the synchronization construct:

```
#pragma omp taskwait
```

3 Measuring Performance

Nebula machine is used to test and measure the performance of the algorithm. Because of the fact that even the serial version of the algorithm worked too fast (see running times from the below chart), no speedup is observed for small sizes of boards. Moreover, a slow down was observed in the algorithm. This was due to paralellization overhead. For small problem sizes, syncronization and scheduling processes of the threads take more time than than the problem itself. Due to this, boards of size 13, 20, 30 are provided from kakuros.com to see whether there is an improvement. As it is seen in the table for problem sizes 20, 30, maximum speedup is achieved by OMP_NUM_THREADS=4.

Num Threads	1	2	4	8	16
Board					
3.1	0.05	0.27	0.44	0.75	6.29
3.2	0.05	0.21	0.39	0.67	8.23
4.1	0.05	0.21	0.41	0.75	36.75
4.2	0.25	0.41	0.56	1.09	33.01
5.1	0.24	0.43	0.64	1.12	35.16
5.2	0.37	0.61	0.79	1.43	48.99
13.1	8.57	9.67	10.19	9.40	50.24
20.1	60.47	33.92	21.22	73.54	103.07
30.1	12408.6	7105.6	4094.78	9199.9	8702.2

Running Time (ms)

Num Threads	1	2	4	8	16
Board					
3.1	x	x	x	x	x
3.2	x	x	x	x	x
4.1	x	x	x	x	x
4.2	x	x	x	x	x
5.1	x	x	x	x	x
5.2	x	x	x	x	x
13.1	x	x	x	x	x
20.1	x	1.78	2.85	x	x
30.1	x	1.75	3.03	1.35	1.42

Speedup

4 Running the Program

The following compiler commands is used in order to compile the program.

g++

g++ kakuro_solver_omp.cpp -fopenmp -O3

clang++

clang++ –std=c++17 -Xpreprocessor -fopenmp kakuro_solver_omp.cpp -lomp -L/usr/local/opt/libomp/lib/ -O3

Sample Execution

./a.out board5_2.kakuro