ICCS 311 Final Project Report

Parallel Sudoku Solver

Worawit Penglam 6281453

Tanapon Techathammanun 6281332

Project Description:

In this project, we aim to create a program to solve the sudoku with a 9*9 grid. We want to utilize and apply our knowledge in the depth first search algorithm (DFS) and backtracking algorithm to solve a given sudoku puzzle. We have created 3 alternative methods:

(1) *Sequential solver*: sequential implementation of DFS and backtracking algorithm
(2) *Parallel solver*: convert our sequential solver into parallel using parallelization techniques in Rust
(3) Rust's *API sudoku solver*: the premade optimal sudoku solver from open API

Goal:

1. Implement all the three methods: sequential, parallel and api to solve a given sudoku puzzle.
2. Make sure that all solvers solve for all the possible solutions.
3. Record the result and compare the runtime of each method.

How the project work:

As mentioned above, we seperated the project into 3 main parts:

(1) In the sequential method we implement the DFS and Backtracking algorithm to make the program solve for all of the possible solutions. But how does this really work? Well, the

program first detects each cell using a variable grid of row and column. If that value in that cell index is 0. The sequential solver will try to plug the number from 1 to 9 into the sudoku grid. If that number is unique i.e. it passes all the three constraints which are (a) unique in that row, (b) unique in that column and (c) unique in its 3 by 3 block, then it will move to the next column to solve it. If it could not find any number which could fit in that cell then it steps back to change the number in the previous cell and so on until all possible numbers are checked hence, we attain all the possible solutions from the input puzzle.

(2) The parallel method, we use the base from the sequential but instead of trying to plug in the number from 1 to 9. We implement it to work in parallel by using into_par_iter(). Moreover, at the part where we check the constraint, we applied rayon::join() to check the rows and columns in parallel.

(3) The last method is api, we use an api called sudoku_solver. This method is used to ensure that the answers from the sequential and parallel method are correct.

Discussion:

We created 3 input puzzles for the sudoku solver to solve and we classified it by difficulty level and the number of possible ways to solve the puzzle. In test case 1, the input is a simple puzzle with only 1 possible solution. The result after compiling the sequential was a little bit faster than the parallel method. So we increased the difficulty for our test case. In test case 2, the result was that parallel runs faster than the sequential as you can see in the picture below. The last test case we made it extremely difficult and had an enormous number of solutions.

TEST 1#:

9,0,6,0,7,0,4,0,3
0,0,0,4,0,0,2,0,0

0,7,0,0,2,3,0,1,0

5,0,0,0,0,0,1,0,0

0,4,0,2,0,8,0,6,0

0,0,3,0,0,0,0,0,5

0,3,0,7,0,0,0,5,0

0,0,7,0,0,5,0,0,0

4,0,5,0,1,0,7,0,8

The result after compile and ran:

```
running parallel sudoku solver
running sequential sudoku solver
running Rust api sudoku solver
Parallel sudoku solver has 2 solutions and it takes: 6.02ms :P
Sequential sudoku solver has 2 solutions and it takes: 4.44ms :P
Api sudoku solver has 2 solutions and it takes: 1.27ms :P
```

TEST 2#:

0,1,7,3,0,0,0,0,5

0,0,0,0,2,0,0,0,0

2,0,0,0,6,7,0,9,1

0,0,2,0,0,0,0,0,9

0,9,6,0,0,0,1,7,0

3,0,0,0,0,0,5,0,0

0,0,0,7,4,0,0,0,8

0,0,5,0,3,0,0,0,4

6,0,3,0,0,9,7,0,0

The result after compile:

```
running parallel sudoku solver
running sequential sudoku solver
running Rust api sudoku solver
Parallel sudoku solver has 1 solutions and it takes: 28.23ms :P
Sequential sudoku solver has 1 solutions and it takes: 37.29ms :P
Api sudoku solver has 1 solutions and it takes: 1.92ms :P
```

TEST 3#:

0,1,7,0,0,0,0,0,5

0,0,0,0,2,0,0,0,0

0,0,0,0,0,7,0,9,1

0,0,2,0,0,0,0,0,9

0,9,6,0,0,0,1,7,0

3,0,0,0,0,0,5,0,0

0,0,0,0,4,0,0,0,8

0,0,5,0,0,0,0,0,4

0,0,0,0,0,0,7,0,0

The result after compile:

```
running parallel sudoku solver
running sequential sudoku solver
running Rust api sudoku solver
Parallel sudoku solver has 727082 solutions and it takes: 158.72s :P
Sequential sudoku solver has 727082 solutions and it takes: 197.81s :P
Api sudoku solver has 727082 solutions and it takes: 58.87s :P
```

Conclusion:

To sum up, our parallel sudoku solver is faster than the sequential method in the case that the puzzle is difficult and has many possible ways to solve. Our parallel solver is not well optimized as we can see from the API method.

Here are the possible points to think about for further optimization:

- The constraint checks could be faster if we do not have to check for all the possible numbers by plugging in the number 1 to 9 in all the constraint checks but instead cross out those which are already repeated.
- Would using fewer threads be faster?
- Over parallelization in lower tree levels could be more costly.

Reference:

https://norvig.com/sudoku.html

https://www.geeksforgeeks.org/sudoku-backtracking-7/

https://www.tutorialspoint.com/introduction-to-backtracking#:~:text=Backtracking%20is%20a%20technique%20based,given%20to%20solve%20the%20problem.

https://medium.com/swlh/sudoku-solver-using-backtracking-in-python-8b0879eb5c2d

https://doc.rust-lang.org/std/collections/struct.HashSet.htm

http://www.afjarvis.staff.shef.ac.uk/sudoku/

https://exercism.org/tracks/rust/exercises/parallel-letter-frequency/solutions/btolfa

https://docs.rs/sudoku-solver/latest/sudoku_solver/
(citation machine should be used we know :P)

We'd like to thank our professor Kanat for continuous support throughout this project.
Have a nice day ^^