Futoshiki Solver - a comparison of Parallel vs Sequential, Limit on OpenMP Tasks, and roll-back mechanism on recursion

Özgün Özerk

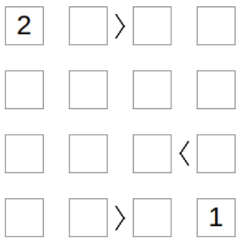
18 April 2020

1 Explanation of the Problem

In this homework, we are given the task of implementing a Futoshiki solver. The definition of the Futoshiki Puzzle and techniques for solving could be found at Futoshiki. The purpose of the assignment was not only to implement a successful solver, but also an efficient solver at the same time. The puzzles will be read from the provided input files, and the filename of the matrix must be passed to the program by command line (argc/argv). The format of the input files will be as follows:

- Each input will begin with an integer n that indicates the size of the matrix.
- Each input will contain a square matrix which cells with values presented by positive numbers and empty cells presented with **-1**'s. The values are separated with white-spaces.
- Each input will have constraints for the puzzle after the matrix. One line for one constraint. Constraints are given in a format that; the value of cell resides in coordinate of first two numbers is bigger than the cell resides in latter two. For example, constraint 1 2 1 3 means that cell at (1, 2) will have bigger value than cell at (1, 3). Therefore an input `input1.txt` will be in the format:

```
4
2 -1 -1 -1
-1 -1 -1 -1
-1 -1 -1 -1
-1 -1 -1 -1
1 2 1 3
3 4 3 3
4 2 4 3
```



2 The Algorithm (Explanation of the functions

Our main function can be seen as "findBestCandidate" function. This function calls "solutionNextStep", which is the pruning function, and after that, calls itself recursively. But before that, giving the simplified algorithm of the pruning might be beneficial for the comprehensive understanding of the code:

Data: focus cell's coordinates, matrix, constraints,
unprocessedCellCount

Result: pruned matrix, true/false depending on dead end

```
if unprocessedCellCount == 0 then
    print the matrix;
else
    prune the matrix with respect to given constraints;
    for every cell do
        if if this cell is empty and has no possible value left then
            return false;
    end
    prune the matrix with respect to the focus cell's column and row;
    for every cell do
        if if this cell is empty and has no possible value left then
            return false;
    end
    unprocessedCellCount–;
    for every cell do
        if if this cell is empty and has only 1 possible value left then
            solutionNextStep(this cell's coordinates);
    end
    return true;
end
```

Algorithm 1: solutionNextStep

Data: matrix, constraints, unprocessedCellCount

Result: void/none

```
if unprocessedCellCount == 0 then
    print the matrix;
else
    Select the cell with the smallest size of possible values, and has the
     most effect on it's neighbours (focus cell) ;
    for every value in the focus cell's possible value list do
        create a new copy of the matrix;
        assign the VALUE to focus cell's coordinates in the new copy;
        create a task;
        *****task region****;
        if solutionNextStep(focus cell's coordinates) then
            findBestCandidate();
        *****end of the task region****;
    end
end
```

Algorithm 2: findBestCandidate

3 Parallelism (necessities and optimizations)

3.1 Necessities

To achieve parallelism, first thing to do is: remove the dependencies between the recursive steps. The OOP version of the Futoshiki solver was operating on the same instance (economical memory usage), and applies roll-back algorithms if a dead end found. In order to being able to apply roll-back algorithm, it also had to store necessary changes and original values in the memory. To remove dependencies, for each branch, another copy of the Futoshiki instance is created, so that no no roll-back was necessary, since the original versions were kept in the stack of the functions (magic of the recursion). A new task will be incumbent upon every branch (Futoshiki instance) that's being generated by the recursive function "findBestCandidate"

3.2 Optimization

The solution could be found with pure brute-forcing, but as the size of instances grow, the time it takes to find a solution grows exponentially. 2 alternatives have been tried:

1. **Find the best candidate by least possible values:** In this one, the algorithm picked the focus cell among the empty-cells with the only criteria being having the least number of possible values. It's performance is unutterably better than the brute-force
2. **Find the best candidate by least possible values and most neighbour effect:** In this one, the same criteria as the above has been followed with an addition. If we found more than 1 cell after applying the above criteria, we check their effects on their neighbours. Thus, selecting the one with the most effect on the Futoshiki Instance. Comparing the performance with the above option, there is not a vast gap between the time it takes to find a solution. Nevertheless, this one proves to be the fastest.

For improving the performance further, 2 major things have been considered (ignoring the small details as passing the arguments with address, etc...)

- **How to store the Futoshiki Instance?**
  First idea was to store the matrix(2D), and store the possible values of each cell in another matrix(3D). Along the progress, storing the size of the possible values as the first index seemed more efficient and meaningful in the second matrix(3D). Thus, another innovation occurred "If we can store an extra value in the 3D matrix, we can store the actual value of the cell as well." Storing also the actual value in the second matrix, eliminated the necessity to store the first matrix(2D), allowing us to reduce total matrices to be copied from 2 to 1 in each recursion.

- **How much parallelism?**
  Keep in mind that, every thread and task increases the overhead. Optimizing the code for dynamic number of threads is carrying significant importance. This optimization in this code is achieved via utilizing "if clauses in OMP tasks". A global integer is keeping the counts of tasks created so far, and another global integer is holding the number of total threads given to the code.

```
1 #pragma omp task if(counter < limit)
2 {
3     counter++;
4     if(solutionNextStep(newMatrix, focusRow, focusCol,
      ↪  constraints, newUnprocessed)){  // prune the
      ↪  matrix and check if creates a dead end
5         findBestCandidate(newMatrix, constraints,
          ↪  newUnprocessed);  // if no cell left to
          ↪  process
6     }
7 }
```

  There is a general misconception about it though. When the "if clause" evaluates to *false*, the task is still created. The difference lies in how the task will be created. When the "if clause" evaluates to *true*, the task is created, and will be run when a thread is empty. We don't and can't know when exactly that task is going to run. On the other

hand, when the "if clause" evaluates to *false*, the thread/task that is creating the new task, suspends it's current work, and starts to run the thread IMMEDIATELY. Lastly, not using the "if clause" is same as using "if(true)". Explanation of the concept becomes more clear with an example:

Suppose only 1 thread is be given to our program. Since there are no multiple threads, only 1 thread will be responsible from all the tasks. And this thread will suspend it's current work (branching), and will start the created task immediately. Resulting in Depth-First-Search.
If there was no "if clause", it would be same as "if(true)". Remember, we can't know when the created tasks are going to run. This scenario is becoming close to a randomized search, being closer to Breadth-First-Search rather than Depth-First-Search.

4 Performance

System and Technology specifications:

CPU:
Intel(R) Xeon(R) CPU E7-4870 v2 @ 2.30GHz **X60**

RAM:
516767 MB

Operating System:
CentOS 6.5

GCC Version:
8.2.0

OpenMP Version:
4.5

Comparison of Parallel vs Sequential

3 input files:

input6.txt, input7.txt, input8.txt, and their instance sizes being 9x9, 10x10, 11x11 respectively.

4 different programs:

S(1): sequential, OOP design with rollback
S(2): sequential, functional design without rollback
Parallel(-L): parallel version, without limit mechanism
Parallel(+L): parallel version, with limit mechanism

Input	*# Threads*		*Time (in seconds)*			
			S(1)	**S(2)**	**Parallel(-L)**	**Parallel(+L)**
input6	1	-	>5 min	>5 min	>5 min	>5 min
	2	Usually	-	-	0.01 - 8	0.11 - 0.14
		Max			41	0.14
	4	Usually			0.08 - 4	0.47 - 0.50
		Max			22	0.59
	8	Usually			0.01 - 0.03	0.03 - 0.12
		Max			29	0.13
	16	Usually			0.03 - 0.07	0.014 - 0.017
		Max			0.5	0.017
	32	Usually			0.07 - 0.12	0.02 - 0.03
		Max			0.14	0.035
input7	1	-	>5 min	>5 min	0.7	>5 min
	2	Usually	-	-	0.004 - 0.14	>5 min
		Max			0.28	>5 min
	4	Usually			0.01 - 0.4	0.01 - 0.5
		Max			25	1
	8	Usually			0.02 - 0.05	0.007 - 0.01
		Max			2	0.02
	16	Usually			0.05 - 0.06	0.017 - 0.025
		Max			0.8	0.28
	32	Usually			0.13	0.15 - 0.34
		Max			0.48	0.37
input8	1	-	>5 min	0.00083	21	0.0009
	2	Usually	-	-	0.004 - 0.007	0.0014
		Max			0.5	0.0016
	4	Usually			0.01 - 0.02	0.003 - 0.005
		Max			0.04	0.006
	8	Usually			0.02 - 0.03	0.007 - 0.011
		Max			0.03	0.022
	16	Usually			0.06 - 0.13	0.01 - 0.02
		Max			0.5	0.02
	32	Usually			0.09 - 0.12	0.019 - 0.038
		Max			0.13	0.04

Discussion: There are lot of things to inspect. First, let's focus on the obvious one: PARALLEL vs SEQUENTIAL. Except for the input3, functional design without the roll-back mechanism, none of the inputs have been solved by sequential programs. Maybe they will take 30 mins, maybe 30 hours... I don't have the patience for waiting, but you are welcome to test if you wish :)

Parallel versions on the other hand, solved these big instances almost all the time under 1 second (lightning fast!). Now for the first interesting part: **What is The Effect of LIMIT on The Parallel Versions?**

Remember, when we do not put limit(if clause), it's the same as we put "if(true)". So in the version of without limit, even if we have not enough threads for that many tasks to be efficiently processed, the tasks are created, and they are waiting to be processed until a thread becomes available and randomly picks them. Imagine you are the task that will yield the solution, but by chance, no threads are picking you for a long time. Additionally, in the meanwhile, other tasks are created, and your chance to getting picked are getting lower and lower. This is the explanation of "Max" values of "Parallel(-L)" differs dramatically from the usual values, whilst "Parallel(+L)"'s "Max" values are impeccably consistent with the usual values. This consistency can be argued with, the branch amount is nearly always the same with the thread count. After generating enough branching, the algorithm turns into Depth-First-Search instead of creating new branches. Same argument is also the reason for the gigantic difference between input2 and input3's runs with a single(1) thread. Let's start with input2:

Although the program only has 1 thread, Parallel(-L) keeps creating new branches, and finds the solution very quickly. This means, the solution is not very deep in the search-tree. But the Parallel(+L) version fails to find the solution even under 5 minutes. Helping us to discover that, the search tree is indeed very deep, but the solution is not in the first branches that are tried by the Parallel(+L). In other words, Parallel(+L) gets lost in the depths of the tree, searching the solution in the wrong place for that instance. Based on input2, we can conclude the exact opposite for input3. The solution is at the branches, that are tried as first from Parallel(+L). So that, Parallel(+L) outperforms Parallel(-L) even with a single thread.

Input	*Size*	*Time (in seconds)*	
		OOP with roll-back	**Functional without roll-back**
input1	4x4	0.000113893	0.000107203
input1_2	4x4	0.000098604	0.000098969
input2	5x5	0.000202812	0.000223842
input2_2	5x5	0.000327289	0.00025174
input3	6x6	0.000753481	0.000703964
input3_2	6x6	0.00578789	0.00506186
input4	7x7		0.00870612
input4_2	7x7	0.0310422	0.0260722
input5	8x8	1.64324	1.43945
inptu5_2	8x8	3.23352	2.80053

4.1 BONUS: Roll-back vs without Roll-back

Discussion: For the sequential code, first one (OOP with roll-back) was written for the sole purpose of being an efficient sequential futoshiki solver. Roll-back mechanism is providing the opportunity the go back to a previous checkpoint, when a dead-end is reached. Hence, opening the possibility for being content with a single instance. On the other side, functional design without the roll-back mechanism is written for parallelism purpose. Roll-back mechanism is replaced with copy instances with the aim of removing dependencies between states. The only difference between the parallel versions and this one is the "pragma omp" statements (since functional design without roll-back is fully parallelizable). The functional design is a bit faster than the OOP design. This performance upgrade is a trade-off between memory and speed as usually. For the bigger instances like 9x9, 10x10 or 11x11, even **terminating** the functional without roll-back after 5 minutes was taking couple of seconds because of the enormous memory usage, whilst OOP with roll-back one's termination was instant.

5 How to Run and Compile

The input file to be read and the actual cpp file should be in the same folder. For compiling, open a terminal in the respective folder, type the command "g++ -std=c++11 fixed.cpp -g -O3 -fopenmp" or wihtout -O3 if you want -O0. Then a binary file will be created (a.out). Type the command "./a.out inputX.txt" (replace x with the the number of the input file. The results should be displayed in the terminal.