CS 531 Homework 2 - Report

Ege Alpay
egealpay@sabanciuniv.edu

April 18, 2020

1 The Code

I have tried to work on alternative1.cpp but I had some difficulties while reading the code and writing copy constructors, so I end up going back to my HW1 code. I had 3 different approaches to make it parallel. Before explaining these designs, I have also changed the part in main function, where solve function was called for the first time.

```
#pragma omp parallel
{
    #pragma omp single
    solve(matrix, constraints, size);
};
```

Listing 1: Calling solve for the first time

1.1 Design 1

```
void solve(int **&matrix, std::vector<std::pair<std::pair<int, int>, std::pair<
    int, int> > > &constraints, int size) {
// Find a empty cell
    std::pair<int, int> emptyCell = findEmptyCell(matrix, size);
    int currentRow = emptyCell.first;
    int currentCol = emptyCell.second;

// If there is no empty cell, then we are done
    if (currentRow == -1 && currentCol == -1) {
        solutionFound = true;
        std::cout << "Can solve: " << solved(matrix, constraints, size) << std::
    endl;
        return;
    }

    for (int value = 1; value <= size && !solutionFound; value++) {
        if (doesSatisfyRules(matrix, currentRow, currentCol, value, constraints,
     size)) {
            matrix[currentRow][currentCol] = value;

            // create copy
            int **matrixNew;
            matrixNew = new int *[size];
            for (int i = 0; i < size; i++) {
```

```
                matrixNew[i] = new int[size];
            }

            int val;
            for (int i = 0; i < size; i++) {
                for (int j = 0; j < size; j++) {
                    matrixNew[i][j] = matrix[i][j];
                }
            }

#pragma omp task
            solve(matrixNew, constraints, size);

            // Invalidate failed
            if (!solutionFound)
                matrix[currentRow][currentCol] = -1;
        }
    }

    return;
}
```

Listing 2: Design 1

Design 1 is very similar to HW1 code, but I have changed return type from boolean to void since returning value in parallel scope is not available in OpenMP. In addition, boolean flag is used to check if any thread has found the solution. In the parallel part, current matrix is copied and every recursive call will be assigned to a thread by creating a task. Since this approach creates task in every call and task creation has an overhead, it is not very efficient.

1.2 Design 2

```
void solve(int **&matrix, std::vector<std::pair<std::pair<int, int>, std::pair<
    int, int> > > &constraints, int size) {
// Find a empty cell
    std::pair<int, int> emptyCell = findEmptyCell(matrix, size);
    int currentRow = emptyCell.first;
    int currentCol = emptyCell.second;

// If there is no empty cell, then we are done
    if (currentRow == -1 && currentCol == -1) {
        solutionFound = true;
        std::cout << "Can solve: " << solved(matrix, constraints, size) << std::
    endl;
        return;
    }

    for (int value = 1; value <= size && !solutionFound; value++) {
        if (doesSatisfyRules(matrix, currentRow, currentCol, value, constraints,
     size)) {
            matrix[currentRow][currentCol] = value;
```

```
17
18              solve(matrix, constraints, size);
19
20              // Invalidate failed
21              if (!solutionFound)
22                  matrix[currentRow][currentCol] = -1;
23          }
24      }
25
26      return;
27  }
28
29  void solveDesign2(int **&matrix, std::vector<std::pair<std::pair<int, int>, std
        ::pair<int, int> > > &constraints, int size) {
30  // Find a empty cell
31      std::pair<int, int> emptyCell = findEmptyCell(matrix, size);
32      int currentRow = emptyCell.first;
33      int currentCol = emptyCell.second;
34
35  // If there is no empty cell, then we are done
36      if (currentRow == -1 && currentCol == -1) {
37          solutionFound = true;
38          std::cout << "Can solve: " << solved(matrix, constraints, size) << std::
        endl;
39          return;
40      }
41
42      for (int value = 1; value <= size && !solutionFound; value++) {
43          if (doesSatisfyRules(matrix, currentRow, currentCol, value, constraints,
         size)) {
44              matrix[currentRow][currentCol] = value;
45
46              if (currentRow < THRESHOLD) {
47                  solveDesign2(matrix, constraints, size);
48              } else {
49                  // create copy
50                  int **matrixNew;
51                  matrixNew = new int *[size];
52                  for (int i = 0; i < size; i++) {
53                      matrixNew[i] = new int[size];
54                  }
55
56                  for (int i = 0; i < size; i++) {
57                      for (int j = 0; j < size; j++) {
58                          matrixNew[i][j] = matrix[i][j];
59                      }
60                  }
61
62  #pragma omp task
63                  solve(matrixNew, constraints, size);
64              }
```

```
65
66              // Invalidate failed
67              if (!solutionFound)
68                  matrix[currentRow][currentCol] = -1;
69          }
70      }
71
72      return;
73  }
```

Listing 3: Design 2

This time we have two solve functions. solveDesign2 is called from main function first. Also we have a threshold value where threshold is equal to (number of rows) / 2. If the current row in the matrix is less than threshold, we will use recursion without creating any task. Else, we will create a copy of the matrix and solve function will be called by creating a new task. With this approach, task creation overhead is reduced, however using parallel programming in the deep level of search tree may create load imbalance between threads since some of them can go deeper in comparison to others.

1.3 Design 3

```
1  void solve(int **&matrix, std::vector<std::pair<std::pair<int, int>, std::pair<
       int, int> > > &constraints, int size) {
2  // Find a empty cell
3      std::pair<int, int> emptyCell = findEmptyCell(matrix, size);
4      int currentRow = emptyCell.first;
5      int currentCol = emptyCell.second;
6
7  // If there is no empty cell, then we are done
8      if (currentRow == -1 && currentCol == -1) {
9          solutionFound = true;
10         std::cout << "Can solve: " << solved(matrix, constraints, size) << std::
       endl;
11         return;
12     }
13
14     for (int value = 1; value <= size && !solutionFound; value++) {
15         if (doesSatisfyRules(matrix, currentRow, currentCol, value, constraints,
        size)) {
16             matrix[currentRow][currentCol] = value;
17
18             solve(matrix, constraints, size);
19
20             // Invalidate failed
21             if (!solutionFound)
22                 matrix[currentRow][currentCol] = -1;
23         }
24     }
25
26     return;
```

```
}

void solveDesign3(int **&matrix, std::vector<std::pair<std::pair<int, int>, std
    ::pair<int, int> > > &constraints, int size) {
// Find a empty cell
    std::pair<int, int> emptyCell = findEmptyCell(matrix, size);
    int currentRow = emptyCell.first;
    int currentCol = emptyCell.second;

// If there is no empty cell, then we are done
    if (currentRow == -1 && currentCol == -1) {
        solutionFound = true;
        std::cout << "Can solve: " << solved(matrix, constraints, size) << std::
    endl;
        return;
    }

    for (int value = 1; value <= size && !solutionFound; value++) {
        if (doesSatisfyRules(matrix, currentRow, currentCol, value, constraints,
     size)) {
            matrix[currentRow][currentCol] = value;

            if (currentRow < THRESHOLD) {
                // create copy
                int **matrixNew;
                matrixNew = new int *[size];
                for (int i = 0; i < size; i++) {
                    matrixNew[i] = new int[size];
                }

                for (int i = 0; i < size; i++) {
                    for (int j = 0; j < size; j++) {
                        matrixNew[i][j] = matrix[i][j];
                    }
                }

#pragma omp task
                solveDesign3(matrixNew, constraints, size);

            } else {
                solve(matrix, constraints, size);
            }

            // Invalidate failed
            if (!solutionFound)
                matrix[currentRow][currentCol] = -1;
        }
    }

    return;
```

```
74 }
```

Listing 4: Design 3

This approach is very similar to Design 2. solveDesign3 is called from main function first. Threshold value is same as Design 2. This time if current row number is less than the threshold, we will call solveDesign3 recursively by creating a new task. So we will use parallel programming before going deep in search tree where task sizes are going to be small.

2 Execution times & Speed-ups

First of all, I would like to share the initial runtime of the where no parallel programming was implemented. In other words, following runtime is obtained by running new inputs with HW1 code implementation:

Table 1: Execution times with Sequential Code

File	Size	Time (Seconds) -O3
input1.txt	9x9	0.03
input2.txt	10x10	1773.49
input3.txt	11x11	0.312

It is very suprising to see input1.txt only takes 0.03 seconds to solve. input2.txt takes forever to find a solution. input3.txt is also much faster than my expectation. Next, you can find the execution times for parallel implementation. I have run each implementation 5 times, marked fastest result by green and slowest by red. In all experiments, sometimes I have observed weird results. For example, in Design 1 on input1.txt with 2 Threads, an execution took 9 seconds to find a solution, another execution took only 0.01 seconds. Maybe task affinity have caused these weird results but I am not sure about the real reason.

Speed-ups are calculated based on the sequential execution time which can be found above and fastest execution time which can be found below. Speed-up stated as ” - ” means that there is no speed-up.

Table 2: Execution times with Design 1

File	Size	Run 1	Run 2	Run 3	Run 4	Run 5	Speed Up
input1 - 1 Thread	9x9	0.25	0.25	0.24	0.25	0.25	-
input1 - 2 Thread	9x9	9.07	3.87	0.01	0.013	0.007	3
input1 - 4 Thread	9x9	0.05	0.17	0.06	0.08	0.017	-
input1 - 8 Thread	9x9	0.13	0.01	0.02	0.02	0.03	3
input1 - 16 Thread	9x9	0.02	0.09	0.12	0.02	0.04	1.5
input2 - 1 Thread	10x10	+300	+300	+300	+300	+300	-
input2 - 2 Thread	10x10	+300	+300	+300	+300	+300	-
input2 - 4 Thread	10x10	103.83	4.59	2.63	128.13	190.38	674.33
input2 - 8 Thread	10x10	7.13	7.20	26.81	106.60	6.80	260.80
input2 - 16 Thread	10x10	261.07	21.99	5.88	14.55	3.44	515.54
input3 - 1 Thread	11x11	0.31	0.317	0.322	0.317	0.319	-
input3 - 2 Thread	11x11	0.695	0.041	0.056	0.104	1.989	7.6
input3 - 4 Thread	11x11	0.073	0.704	0.949	0.093	0.351	4.27
input3 - 8 Thread	11x11	0.493	0.077	0.05	0.077	0.098	4.05
input3 - 16 Thread	11x11	0.359	0.228	0.136	0.244	0.263	2.29

Implementation of Design 1 was easy but since task creation has overhead and creating copy of the matrix has $O(n^2)$ complexity, performance improvement was not satisfying. Input1 and Input3 got faster with 2 Threads but increasing thread number did not help after 2 Threads. Input2 was the hardest input for my algorithm, 2 Threads did not improve the performance but using 4 or more Threads did help actually.

Table 3: Execution times with Design 2

File	Size	Run 1	Run 2	Run 3	Run 4	Run 5	Speed Up
input1 - 1 Thread	9x9	0.055	0.055	0.055	0.056	0.056	-
input1 - 2 Thread	9x9	0.033	0.032	0.032	0.032	0.032	1
input1 - 4 Thread	9x9	0.032	0.034	0.032	0.032	0.033	1
input1 - 8 Thread	9x9	0.032	0.032	0.032	0.032	0.032	1
input1 - 16 Thread	9x9	0.032	0.032	0.033	0.032	0.032	1
input2 - 1 Thread	10x10	+300	+300	+300	+300	+300	-
input2 - 2 Thread	10x10	+300	+300	+300	+300	+300	-
input2 - 4 Thread	10x10	+300	+300	+300	+300	+300	-
input2 - 8 Thread	10x10	227.45	227.30	227.33	227.54	227.26	7.80
input2 - 16 Thread	10x10	115.20	114.87	113.98	114.07	112.11	15.81
input3 - 1 Thread	11x11	0.326	0.327	0.328	0.327	0.327	-
input3 - 2 Thread	11x11	0.327	0.327	0.328	0.328	0.329	-
input3 - 4 Thread	11x11	0.323	0.323	0.323	0.323	0.323	-
input3 - 8 Thread	11x11	0.324	0.323	0.323	0.323	0.323	-
input3 - 16 Thread	11x11	0.324	0.323	0.322	0.323	0.323	-

In Design 2, a THRESHOLD was defined to control task generation. While current row number is less than the THRESHOLD, all the recursive calls are made by a single thread without creating the copy of the matrix. After reaching the THRESHOLD, new recursive calls are made by creating a task by copying the matrix. With this Design, task creation overhead was controlled and memory usage was decreased. Although since task generation was occurred in deep levels of the search tree, load imbalance can be observed. Interestingly, it was running slower than Design 1. In addition, observation of performance improvement can only be seen for Input2.

Table 4: Execution times with Design 3

File	Size	Run 1	Run 2	Run 3	Run 4	Run 5	Speed Up
input1 - 1 Thread	9x9	0.06	0.06	0.06	0.06	0.06	-
input1 - 2 Thread	9x9	32.11	0.009	0.004	0.08	0.06	7.5
input1 - 4 Thread	9x9	0.003	5.57	0.021	0.005	0.031	10
input1 - 8 Thread	9x9	0.140	0.002	0.042	0.128	0.006	15
input1 - 16 Thread	9x9	0.010	0.020	0.070	0.017	0.026	3
input2 - 1 Thread	10x10	+300	+300	+300	+300	+300	-
input2 - 2 Thread	10x10	20.14	6.87	6.72	5.18	21.30	342.37
input2 - 4 Thread	10x10	1.11	6.00	1.11	6.26	14.38	1597.73
input2 - 8 Thread	10x10	4.59	7.84	5.35	22.63	1.81	979.82
input2 - 16 Thread	10x10	10.96	6.89	1.12	7.22	0.33	5374.21
input3 - 1 Thread	11x11	0.326	0.327	0.328	0.327	0.327	-
input3 - 2 Thread	11x11	0.008	0.008	0.008	0.009	2.07	39
input3 - 4 Thread	11x11	0.014	0.016	0.109	0.028	0.021	22.28
input3 - 8 Thread	11x11	0.027	0.013	0.019	0.027	0.020	24
input3 - 16 Thread	11x11	0.053	0.024	0.021	0.032	0.036	14.85

Design 3 looks very similar like Design 2 but this time, if current row number is less than the THRESHOLD, recursive call was done by creating a new task. With this approach, new tasks were created without going deep in the search space where tasks had less load in comparison to Design 2. In overall, performance is better in comparison to Design 1 and Design 2.

Consequently, please consider Design 3 as the real results for this homework. "futoshiki_hw2_design3.cpp" includes the code for Design 3.

3 Compile and Run

For development, I have used CLion IDE to write the code. For compilation and testing the implementation, I have used gcc 9.2.0 on my Linux Mint machine. On Gandalf, gcc 8.2.0 was used.

Following command was used to run the code on Gandalf:

```
module load gcc/8.2.0
gcc -fopenmp hw1.cpp -lstdc++
./a.out inputs/input1.txt

```

Listing 5: Run command on Gandalf