TÉCNICO
LISBOA

Parallel and Distributed Computing

Report 1st Project

Parallel Sudoku using OpenMP

Professor Luís Guerra e Silva
2^{nd} Semester – 2017/2018
08/04/2018

Group 9:

81731 – Catarina Aleixo
81074 – Manuel Reis
81216 - Bernardo Amaral

I. Serial implementation

The program that was developed receives a file as input and first fills a game board that corresponds to the Sudoku given. There is a structure called Board that contains an unidimensional array that represents the Sudoku game board, the square size of n and the size n, where the square size is given by the first line that is read from the given file. This structure also contains three other vectors that are masks for the rows, columns and boxes of the Sudoku game board, that are used to check if a given number is valid on a certain position when the Sudoku is being solved. The game board is represented by the unidimensional array of type Cell that is another structure that represents each cell of the board and contains two integers: value and fixed. The value is a number between 1 and n for the fixed cells and 0 for the unfixed ones, fixed is a flag that states if a certain cell has a fixed value (for instance, the values different from 0 that are read from the file are fixed). After filling the game board, the function *bruteForce()* is called.

This is where the implementation of the Sudoku solver is done. It was implemented a simple search algorithm similar to the DFS, using backtracking. In the *bruteForce()* function, the program will search for the unfixed cells and tries to assign a possible value for each of these cells (between 1 and n) calling the *makeGuess()* function and there it will be tested if there's no conflict in the corresponding row, column and box using the *checkValidityMasks()* function. When there's no possible values for a certain cell, then the function will backtrack to the last unfixed cell and it will assign the next valid value for that cell. If the backtracking reaches the first cell of the game board and there's no more possible values to test, that means that there's no solution for that Sudoku.

II. Parallel implementation using OpenMP

- **Decomposition:**

In the parallel implementation, it will also start by filling the game board and then the function *solver()* is called. In this function, the program will look for the first unfixed cell and then starts the parallel implementation with a ***#pragma omp parallel*** directive. First, it will check which are the possible values for that first unfixed cell using a ***#pragma omp parallel for*** with a ***schedule(dynamic)*** clause. For each possible value, it will be assigned a new board which is a copy of the original board with that cell filled with the possibility. After that, it will be generated tasks with the directive ***#pragma omp task***. These tasks will perform the brute force serial algorithm with some changes, having in account the load balancing and the data races. The function called for each task is the *taskBruteForce()* which receives a game board, a start index, the number of threads and a variable that is called threshold. Each thread will perform the *taskBruteForce()* function in an independent game board and it will start in the first unfixed cell that is not filled for that game board.

- **Load balancing:**

In order to keep all threads occupied, when a certain thread is performing the algorithm, in its board, it will check periodically, after filling square size unfixed cells, if there are idle threads, comparing the number of active threads with the number of threads. If both conditions are met then it will search for a valid number, calling the function *makeGuess()* for that cell and if there's at least one, it will make a copy of its board with that value assigned to that cell and will generate a task with the directive ***#pragma omp task***, which consists in calling the *tasksBruteForce()* function starting in the next cell. Then the thread will continue searching for the next valid number for that cell, if there's one, otherwise it will backtrack to the last unfixed cell filled. Here it has a difference to the serial algorithm since the backtracking it will be performed not at the first cell of the game board, but only until the index where that board started when it took a task from the pool. The threshold variable is used to keep track of the fixed cells of the game board, since each thread will check if it has to distribute work after square size unfixed cells, so the program has to keep track of the fixed cells for each thread in each game board in order to be sure that certainly square size unfixed cells were filled. This distribution of work is done only if the number of threads is greater than one.

- **Synchronization concerns:**

If some thread finds a solution, it's used a global variable called *END* to abort other threads. If there's no more tasks and the *END* variable is with state *FALSE* means that no solution was found. However, if threads find a solution at the same time, only one will change the *END* variable and print the solution, since it was used a ***#pragma omp critical*** to lock the access. It's also used a variable to keep track of the active threads. Every time that a task is assigned, this variable is incremented and when a thread finds that in that game board there's no solution, decrements this variable. Once this is a global variable for all threads, the increments and decrements are performed after a ***#pragma omp critical*** directive, since this variable will implicitly determine whether work needs to be created or not. Without data access control, could happen both, an over or/and an underflow of tasks created.

III. Experimental and Performance results

The running time is expected to be always better in the parallel approach when a given Sudoku has no solution, since all threads will have to search the entire space, when in serial, all the work would be in charge of one thread. Since masks are being used, verifying if a certain value keeps the board valid for a given index is O(1). Particularly in the 16x16.txt file, it was seen that the execution time tends to be different from execution to execution, since the parallel approach is dynamic, this means that a possible easier task can be performed in last place and, since harder tasks can also have a solution, a best execution time isn't guaranteed if all threads are solving time consuming tasks. So, these execution times and speed ups presented in the following tables can vary, since the given Sudoku puzzles have more than one solution which means that it's a fair comparison only if the serial solution is the same as the parallel solution, which not happens in the majority of times. This means that, although these tables reflect the differences between serial and parallel implementations, they aren't always about the same solution, which turns some results worst in parallel than in serial. It's important to refer that in 16x16 file, it can have speed down if finds another solution, which can be seen on table 1 (both 4 and 8 threads with two different times depending on the solution found).

File	Nr of threads	Running time (m:s)	Speed Up
9x9	1	0 : 0,138	1
	2	0 : 0,006	23
	4	0 : 0,006	23
	8	0 : 0,006	23
9x9-nosol	1	0 : 16,890	1,01
	2	0 : 10,724	1,60
	4	0 : 6,481	2,65
	8	0 : 6,779	2,54
16x16	1	0 : 15,422	1,02
	2	0 : 15,812	0,99
	4	0 : 0,089 / 0 : 16,603	177,31 / 0,95
	8	0 : 0,140 / 0 : 26,348	112,72 / 0,60
16x16-zeros	1	0 : 0,006	0,83
	2	0 : 0,006	0,83
	4	0 : 0,007	0,71
	8	0 : 0,009	0,56
16x16-nosol	1	19 : 44,46	0,99
	2	19 : 23,54	1
	4	6 : 07,77	3,17
	8	4 : 53,29	3,98

Table 1 – Parallel performance

Note: these running times were done in a 2.2GHz quad-core Intel Core i7 processor

File	Running time (m:s)
9x9	0 : 0,138
9x9-nosol	0 : 17,191
16x16	0 : 15,781
16x16-zeros	0 : 0,005
16x16-nosol	19 : 26,91

Table 2 – Serial performance