Parallel Sudoku @ OpenMP

Group 10:
Artur Esteves - 91603
Ricardo Morais - 91064
Luís Fonseca - 79770

1 Serial solution

The algorithm implemented to solve a n by n sudoku grid is a backtracking algorithm which visits the empty cells from top to bottom and assigns a number sequentially, i.e., from 1 to n. It backtracks when no number can be placed in an empty cell due to the sudoku rules. This kind of brute force algorithm always gives a solution if the puzzle is valid. A big disadvantage is that it takes a lot of time to check if the puzzle has a solution or is invalid.In the worst case scenario, the algorithm has $\mathcal{O}(n^k)$ complexity where n is the size of the sudoku and k is the number of empty cells.

2 Parallel solution

Our parallel solution is done using openMP tasks. We defined each task as a recursive depth-first search on a subtree of the original search tree, and we added a branch that executes a task whenever a thread is available to pick it up. Every task creates a copy of the puzzle state (cell position, value, puzzle) at the moment of the task creation in order to run independently. At each level of the recursion, a parent task waits for the child tasks to be completed allowing the algorithm to backtrack. This approach is similar to an approach with a **pragma omp parallel for** directive because once this algorithm starts, n tasks will be created and distributed to every thread, those tasks will behave as independent depth-first searches.

3 Experimental Times

Sample	Searched states	Ellapsed time	States per second
9x9	361977	0.180000 (s)	2010983
9x9-nosol	48360826	22.890000 (s)	2112749
16x16-zeros	257	0.004000 (s)	64250
16x16	28322456	37.028000 (s)	764892

Table 1: Execution results of the series implementation

Sample	Threads	Searched states	Ellapsed time	States per second	Speedup
9x9	1	361977	0.213000 (s)	1699422	0.845
9x9	2	362001	0.200000 (s)	1810005	0.9
9x9	4	901	0.003000 (s)	300333	60
9x9	8	1090	0.008000 (s)	136250	22.5
9x9-nosol	1	48360826	26.332000 (s)	1836580	0.869
9x9-nosol	2	48360826	25.547000 (s)	1893013	0.895
9x9-nosol	4	47872228	15.212000 (s)	3147004	1.504
9x9-nosol	8	47446588	15.008000 (s)	3161419	1.525
16x16-zeros	1	257	0.001000 (s)	257000	4
16x16-zeros	2	497	0.019000 (s)	26157	0.210
16x16-zeros	4	586	0.019000 (s)	30842	0.210
16x16-zeros	8	1575	0.022000 (s)	71590	0.181
16x16	1	28322456	33.906000 (s)	835322	1.092
16x16	2	28322467	32.227000 (s)	878842	1.148
16x16	4	82535234	38.522000 (s)	2142547	0.961
16x16	8	104470130	44.991000 (s)	2322022	0.823

Table 2: Execution results of the parallel implementation

The 16x16-nosol sample never returns in both the serial and parallel cases because the state space is too big for a simple depth-first search without optimizations.

4 Conclusions

The speedup is dependent on the time it takes for the solution to be found. This measurement does not characterize the parallelization effort on the depth-first search algorithm since using a depth-first search, the algorithm can reach a solution very fast if it is lucky enough, making the effort of the parallel threads useless. A better way to check if the algorithm is doing its job is by measuring the number of states the algorithm has searched as well as the ellapsed time of the execution. Since a single sample is not representative of the whole improvement of the parallelization and since the objective of this project is not to focus on the heuristics of the algorithm(only to focus on the parallelization of the serial implementation), we can say that even if the speedup is not demonstrated in the sample puzzles supplied, the work that is being done by the threads is significantly more than that of its series counter-part. Of course this leaves us with a problem, which is to assess if it is worth to parallelize? One advantage of the parallelization of the depth-first search is that the parallel version can search states faster, this is particularly useful when we want to search the entire search space to find that no solutions are available. Another empirical point in favor of the parallelization is that the more depth-first searches that are done in parallel, the more likely it is to reach a solution faster. So if we disregard the overhead of starting and synchronizing the threads and start e.g. 8 depth-first searches running each in one of 8 threads in parallel, we get a higher probability of finding the solution in less time than with 1 depth-first search and 1 thread. The main disadvantage of the depth-first search appears when we want to parallelize it. There are a lot of dependency concerns that can only be mitigated by performing costly copies of the intermediate puzzle states.