Cristian Dávila
Enrique González

Deliverables

You should deliver a compressed tar file (GZ or ZIP) with the PDF that contains a report for the work done in this session and the parallel OpenMP code, including Extrae instrumentation (which should be activated when compiling with -D EXTRAE in the Makefile). The report should have an appropriate structure, using sections to cover the different aspects that you want to describe, and make adequate use of tables and figures in order to present the information in the most effective way. In this report you should address, at least, the following issues:

1. Explain how the program implements the recursive exploration and the main data structures that are used.

Program:
From a board, we check the possibilities of each box recursively.
For each call to Solve, see if this is the last box, if so, the number of solutions found. If the first solution is found, the shows on screen.
If you're not in the last box, check if this box has an assigned value, if it has no value, it checks all possible values, if you find a valid one, set ‘solve’ to 1, and continue looking values.

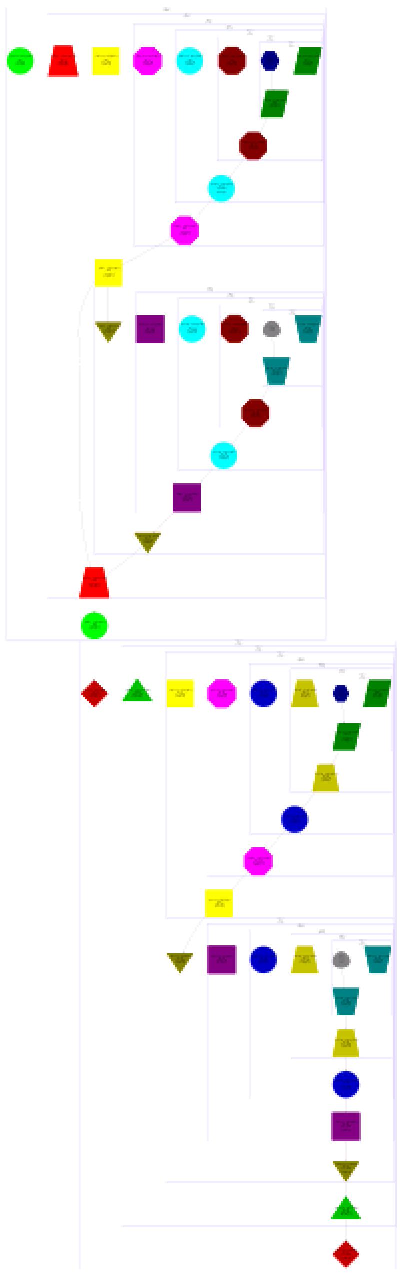
Data structure:

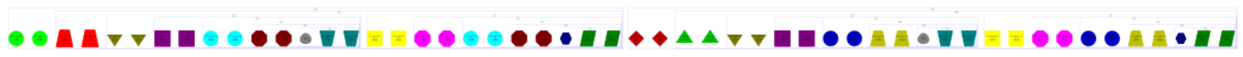
- int *g: current copy of the board
- int *first_solution: structure where the first solution found is saved
- allGuesses[size*size]: vector where we store all possible values per cell

2. From the analysis with Tareador, name the variables and accesses to them that cause the serialization of the tasks in the initial task decomposition that we provide. Include the final task graphthat is generated by Tareador before and after filtering these variables.

Variables:

- int *g
- int *first_solution
- int num_solutions





As we can see, when we have disabled the previously mentioned variables, theorically we'll can fully parallelize the rest of the execution.

3. Explain any changes done in the original source code in order to make it amenable for parallelization.

To ensure the parallelization, we need that the variables that were causing serialization are private access. We've added a #pragma omp single to increase ‘num_solutions’ , exclusion zone with #pragma omp critical for first_solution, and a copy of ‘g’ in #pragma omp task inside the for.

4. In order to reduce the overheads introduced by the parallelization, did you needed to control the recursivity level for which tasks are generated? How?

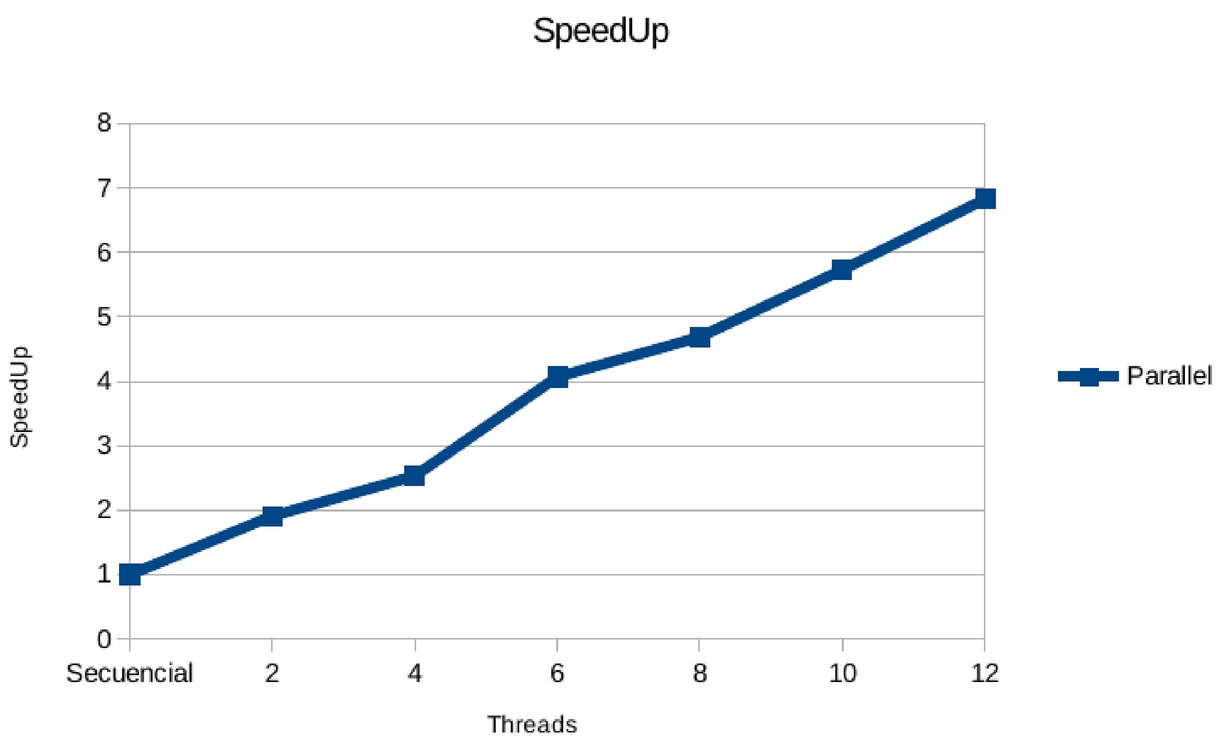
Yes. We've added a parameter to control the recursion which we descended. From the value of this parameter within the function if we decided to create a new task, or if enough number of tasks we have.

5. The basic architecture of the system you have used for the experimental evaluation and the input files used. Comment about the size of the puzzle, number of solutions found, same solution or not reported by the serial and parallel code, …

The performance of the parallelization strategy for N-Sudoku has been analyzed on a multiprocessor architecture with L1 cache of 64k, L2 cache of 256k and L3 cache 12288k, 6 core and 2 thread per core.
For all the performance results we have used an input puzzle of size 3. We have found 730333 solutions. The two strategies have the same solution, because the two codes save the first solution.

6. Comment about the execution time and speed–up, with respect to the original sequential code, for the parallel execution with up to 12 processors. Reason here about the numbers reported and how did you decide about the most appropriate maximum recursion depth for which tasks are generated.

Cores	Sequential	Parallel	SpeedUp
2	8.758093 s	4.584188 s	1,9105
4	8.758093 s	3.459325 s	2,5317
6	8.758093 s	2.151984 s	4,0697
8	8.758093 s	1.867040 s	4,6908
10	8.758093 s	1.527209 s	5,7347
12	8.758093 s	1.282898 s	6,8268



We can see that by increasing the number of CPUs the speed-up enhances too. It's because the threads won't wait until the other threads have finished.
To decide about the most appropriate maximum recursion depth, we have done some executions varying the maximum recursion depth for which tasks are generated. In that case the recursion depth remains constant since 4 to 25, and it's in this point where the overhead of creating tasks is greater that the improvement we get to parallelize.