PFP Lab B: Sudoku—in parallel

John Hughes

Introduction

In this lab you will solve Sudoku[1] puzzles—but you don't need to write (another) Sudoku solver. You will find one in sudoku.erl. There are a number of problems, of varying difficulty, in problems.txt; problems are represented as Erlang data-structures, like this:

```
{wildcat,
 [[0,0,0,2,6,0,7,0,1],
  [6,8,0,0,7,0,0,9,0],
  [1,9,0,0,0,4,5,0,0],
  [8,2,0,1,0,0,0,4,0],
  [0,0,4,6,0,2,9,0,0],
  [0,5,0,0,0,3,0,2,8],
  [0,0,9,3,0,0,0,7,4],
  [0,4,0,0,5,0,0,3,6],
  [7,0,3,0,1,8,0,0,0]]}.
```

Each problem is a list of nine lists, each of nine elements, where elements from 1 to 9 represent fixed digits in the problem, and zeroes represent spaces where a digit is to be filled in. Calling

```
sudoku:benchmarks().
```

solves each puzzle 100 times, and measures both the time to solve each puzzle (in ms), and the time to run all the benchmarks (in µs).

```
4> sudoku:benchmarks().
{80425000,
 [{wildcat,0.68},
  {diabolical,65.18},
  {vegard_hanssen,144.55},
  {challenge,9.93},
  {challenge1,521.99},
  {extreme,13.13},
  {seventeen,48.79}]}
```

This is expected to take around a minute.

Your task is to speed up the benchmarks using parallelism.

Running benchmarks in parallel

The most obvious way to speed up the `benchmarks/0` function is to solve the different puzzles in parallel. Implement this idea, and measure the speedup you obtain for `benchmarks/0`. Submit your modified source code, the output of the sequential and the parallel benchmark. Compute the

[1] If, by some mischance, you are still unfamiliar with Sudoku, look up the rules in Wikipedia.

speedup you obtain, and report how many cores and what type of machine you used. What can you say about the results?

Note that you should *not* parallelize the `repeat/1` function, which solves each puzzle a large number of times. The reason for `repeat/1` is just to make your benchmark run a bit longer, so that you can make more accurate measurements and get more accurate graphs—if you parallelize `repeat`, then you defeat the purpose of the exercise. You may change the number of repetitions of each solving (?EXECUTIONS) to suit the machine you are running on: the benchmarks should run for long enough that you can measure their time accurately, but not so long that you spend a lot of time waiting for them to finish.

Understanding the solver

Because the puzzles take a varying length of time to solve, and we cannot tell in advance which puzzles will be the slowest to solve, then just running a sequential solver several times in parallel will not give the best speedup. Rather, we should make the solver itself parallel. To do so, we must understand how it works.

Puzzle representations

The problems supplied as input are matrices containing zeroes in the unknown positions, but they are converted to ”partial solutions”, by `fill/1`, in which unknown elements are replaced by a *list* of possible values. Think of this as ”filling in each square with the values that might appear there”. The solver itself operates on these partial solutions, gradually removing elements from the lists of possible values, until each list has only a single element—at which point the value in the square is known, and the puzzle is solved.

Refining partial solutions

Given a partial solution, one way to refine it is to remove from each set of possible values, all the values already known to occur in the same row, the same column, and the same block. If no values remain in any square, then the puzzle cannot be solved; if there is exactly one value remaining in a set of values, then that must be the value in that square. The function `refine/1` applies this idea repeatedly to a partial solution, until no more values can be removed from any set by this method. This method alone is sufficient to solve easy puzzles such as wildcat:

```
8> sudoku:refine(sudoku:fill(Wildcat)).
[[4,3,5,2,6,9,7,8,1],
 [6,8,2,5,7,1,4,9,3],
 [1,9,7,8,3,4,5,6,2],
 [8,2,6,1,9,5,3,4,7],
 [3,7,4,6,8,2,9,1,5],
 [9,5,1,7,4,3,6,2,8],
 [5,1,9,3,2,6,8,7,4],
 [2,4,8,9,5,7,1,3,6],
 [7,6,3,4,1,8,2,5,9]]
```

Harder problems such as diabolical cannot be completely solved by this method, so the result of `refine/1` still contains unknown squares.

```
10> sudoku:refine(sudoku:fill(Diabolical)).
[[[1,4,7,9],2,[1,3,4,7],6,[1,3,5,7],8,[1,3,4,9],[4,5],[1,5,9]],
 [5,8,[1,3,4,6],[1,2],[1,3],9,7,[2,4,6],[1,2]],
 ...]
```

However, for each element, we know what the range of possible values are. For example, the top left element of diabolical must be 1, 4, 7 or 9.

Guessing

Once we have drawn all the inferences we can by refinement, we simply pick a square, and guess what the value in it might be. We have a list of possible values in the square, so we can just guess each one of those in turn, and see if we can solve the resulting puzzle recursively. If we fail to solve the puzzle for our first guess, then we try the second guess instead, and so on. If we can't solve the puzzle for *any* guessed value, then the puzzle as a whole is insoluble.

Which square should we pick, to guess the value of? Well, since we know how many guesses we will have to try for each square, then we can pick one of the squares with the fewest possible guesses, to keep the cost of the search as low as possible. The function `guess/1` chooses a square in a matrix to guess by this method. The function `guesses/1` returns a list of resulting matrices, after refinement, with the *easiest* matrix first. (It's possible, of course, that one guessed value for a square leads to much more helpful refinement of other squares than another. It makes sense to try the helpful guesses first!).

Finally, the function `solve_refined/1` applies the whole recursive search algorithm to solve a puzzle completely, raising an exception if it is not solvable. We assume that the matrix given to `solve_refined/1` has already been refined, because this is the case in the recursive calls; the top-level function `solve/1` just refines its argument and then calls `solve_refined/1`.

Read the code in sudoku.erl, try it out, and make sure you understand it.

Parallelizing the solver

Your goal now is to speed up the solution of *one* puzzle using parallelism. There are several opportunities for parallelism in the solver algorithm above. For example,

- When refining the rows of a matrix, we could refine all the rows in parallel
- We could explore the different possible guess values for the guessed square in parallel—but note that only *one* solution is needed, so there is a risk of wasting work looking for alternative solutions in parallel, and thereby finding the first solution *slower* than a depth first search would!

These are two possibilities: there may well be more. Experiment with these methods, and measure the speedups (or slowdowns!) you obtain. Use the *sequential* version of the `benchmark/0` function to measure your performance in this part of the exercise, so that the times you measure are the times to solve one puzzle with all your available cores—the only parallelism you use should be inside the solver itself.

In this exercise, the intention is that you work with the parallelism primitives provided by Erlang, so you should not use ready-built components such as the generic server, or other open source code taken from the web; if you need a particular kind of component, build one of your own using Erlang's language primitives.

My solution solves the benchmark problems around three times faster than the sequential original, running on an Intel Core i7-6700K SkyLake quad core processor with hyperthreading.

Submit your parallelized code, together with a brief description of the methods you used to parallelize it. Include the output of running the parallel benchmarks, together with a description of

the machine used to run them, and a copy of the output of the sequential benchmark on the same machine. Compute the speedup for solving each puzzle, and the geometric mean of all your speedups. The best speedup wins bragging rights!

Enjoy!