Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής & Υπολογιστών
`http://courses.softlab.ntua.gr/pl2/`

Γλώσσες Προγραμματισμού ΙΙ

Αν δεν αναφέρεται διαφορετικά, οι ασκήσεις πρέπει να παραδίδονται στους διδάσκοντες σε ηλεκτρονική μορφή μέσω του συνεργατικού συστήματος ηλεκτρονικής μάθησης `moodle.softlab.ntua.gr`. Η προθεσμία παράδοσης θα τηρείται αυστηρά. Έχετε δικαίωμα να καθυστερήσετε το πολύ μία άσκηση.

Άσκηση 3   Erlang και Παραλληλία

Προθεσμία παράδοσης: 22/11/2015

Ο σκοπός της συγκεκριμένης άσκησης είναι να σας εξοικειώσει με την παράλληλη εκτέλεση Erlang προγραμμάτων. Θα εξετάσουμε ένα πρόγραμμα επίλυσης sudoku. Δε χρειάζεται να γράψετε το πρόγραμμα! Ένα τέτοιο πρόγραμμα υπάρχει στην ιστοσελίδα του μαθήματος ως υλικό σχετικό με τη συγκεκριμένη άσκηση.[1] Το μεγαλύτερο μέρος της υπόλοιπης εκφώνησης είναι γραμμένο στα Αγγλικά διότι η περιγραφή του αλγόριθμου επίλυσης έχει ανακυκλωθεί από αλλού.

Αναπαράσταση των προβλημάτων και περιγραφή της λύσης

The problem puzzles are represented as Erlang tuples; one of them is shown in Σχήμα 1α□.

```
{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]]}.
```

(α□) The representation of a puzzle

```
2> sudoku:benchmarks().
{19166285,
 [{wildcat,0.3106666666666667},
  {diabolical,23.09654761904762},
  {satanic,44.854238095238095},
  {challenge,2.922690476190476},
  {real_challenge,182.32959523809524},
  {extreme,3.9874523809523805},
  {more_extreme,6.355666666666667},
  {seventeen,14.6585},
  {excruciating,2.271095238095238},
  {difficult,1.4930238095238095},
  {very_difficult,13.206880952380953},
  {climbing_everest,138.7065},
  {absurd,11.865095238095238},
  {hard,42.24604761904762},
  {empty,10.281642857142858}]}
```

(β□) Example run of the `benchmarks` function

Σχήμα 1: The puzzle representation and the benchmarking output

Each puzzle is a list of nine lists, each of nine elements, where entries 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 42 times (this is controlled by a macro called `EXECUTIONS`) and reports both the time to solve all of the puzzles (in μs) and each of them (in ms) as shown in Σχήμα 1β□. Your task is to speed up their solution using parallelism.

[1] Το πρόγραμμα βρίσκεται σε ένα αρχείο `sudoku.erl`. Η ιστοσελίδα επίσης περιλαμβάνει ένα αρχείο κειμένου `sudoku_problems.txt` με κάποια sudoku προβλήματα διαφορετικής δυσκολίας, ένα αρχείο `sudoku_solutions.txt` με τις λύσεις τους και ένα `Makefile` που μπορεί να κάνει την ενασχόλησή σας με την άσκηση λίγο πιο εύκολη.

Running the benchmarks in parallel

The most obvious way to speed up the execution of the program is to solve the different sudoku puzzles in parallel, each on a separate Erlang process. Implement this idea and measure the speedup you obtain for running `sudoku:par_benchmarks/0`, the new exported function you will introduce for this purpose. Use the percept tool to visualize the parallelism you get, and submit your modified source code, the output of the sequential and the parallel benchmark, and a percept graph showing the parallelism you managed to obtain. Make sure to report the characteristics of the machine (processor model, number of cores, etc.) and the Erlang/OTP version you used to run your experiments. Also specify any command-line options you may have supplied to the `erl` command.

Note that you should not parallelize the `repeat/1` function, which solves each puzzle a 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 (the `EXECUTIONS` macro) 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. If you did modify this macro, make sure to also report the number of iterations you chose and why.

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 slow ones, then just running a sequential solver several times in parallel will not give the best speedup. Rather, we also 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 "making a note 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.

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`:

```
3> c(sudoku, export_all).
{ok,sudoku}
4> {ok, Puzzles} = file:consult("sudoku_problems.txt"),
4> [Wildcat] = [Puzzle || {wildcat, Puzzle} <- Puzzles],
4> 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.

```
5> [Diabolical] = [Puzzle || {diabolical, Puzzle} <- Puzzles],
5> 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]],
 [[1,7,9],[1,3,6,9],[1,3,6,7],[1,2,5,7],4,...
```

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 cannot 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 is 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, returning the atom `no_solution` 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:

- When refining the rows of a matrix, we could refine all the rows in parallel.
- We could refine the rows, columns, and blocks of a matrix in parallel, and then take the intersection of the results.
- We could explore the different possible guess values for the guessed square in parallel.

These are just three possibilities; there may well be more. Experiment with these methods and measure the speedups you obtain. Use the time of the sequential version of the `benchmark/0` function to measure your speedups in this part of the assignment, 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.

Οδηγίες υποβολής

Για τη συγκεκριμένη άσκηση θα πρέπει να παραδώσετε:

1. Τον κώδικα του προγράμματός σας. Το πρόγραμμά σας θα πρέπει να ορίζει τρεις exported συναρτήσεις `benchmarks_parallel/0`, `solve_all_parallel/0`, και `solve_parallel/1`, αντίστοιχες με αυτές του αρχικού προγράμματος.

2. Μια σύντομη περιγραφή των μεθόδων που χρησιμοποιήσατε για την παραλληλοποίηση του προγράμματος και δυσκολίες που τυχόν συναντήσατε.
3. Screenshots από το percept και από το τρέξιμο του προγράμματός σας σε κάποιο πολυπύρηνο μηχάνημα, καθώς και υπολογισμό των speedups που καταφέρατε να επιτύχετε για κάθε puzzle όπως και συνολικά στο συγκεκριμένο μηχάνημα (και μια περιγραφή της αρχιτεκτονικής του, αν δεν πρόκειται για κάποιο μηχάνημα του SoftLab).
4. Ότι άλλο νομίζετε ότι πρέπει να αναφέρετε (π.χ. command-line arguments που χρησιμοποιήσατε) ή/και πράγματα που προσπαθήσατε να κάνετε αλλά τελικά δε σας δούλεψαν και γιατί.