Assignment on Actors

Language Abstractions for Concurrent and Parallel Programming 2017

Due Date: Friday, 8 December 2017, 18:00:00

0 Getting Started

In https://www.it.uu.se/datordrift/maskinpark/linux you can find a list of the departmental machines that run Ubuntu. They all have installed a very recent version of Erlang/OTP (19.x) and, most likely, more cores than your home machine. But still, you will most probably also want to install Erlang/OTP on your home machine. You can do so either by following the instructions at http://www.erlang.org and getting a tarball from there or, if you have a Linux machine with most needed packages already installed, you can do that with the following commands:

```
git clone https://github.com/erlang/otp.git
cd otp
./otp_build autoconf
./configure
make -j 4
```

This will build the Erlang/OTP system and leave the `erl` command in `bin/erl`. Make sure you use this `erl` command from now on. (Hint: modify the `PATH` variable of your shell.) Alternatively, you can issue a `make install` command at this point. In either case, you are now ready to fire up the Erlang shell and go through the set of slides on Erlang from the course's homepage (on the student portal).

On a Mac with homebrew installed, you can type `brew install erlang`.

1 Warming Up

The lecture slides contain the code of a QuickSort function (`qsort/1`) in Erlang. To get yourself familiar with Erlang programming, especially in case you have not used Erlang or another functional language before, we suggest you implement some other sorting algorithm, e.g. `mergesort/1`.

Hint: you need to implement three simple functions for this task: a function `split/1` that traverses a list and splits it into two lists of equal or about equal length, a function `merge/2` that merges two sorted lists into a single sorted list, and `mergesort/1` itself. We also suggest that you test each of your functions separately, as you develop them. Also, that you try to test and measure the performance of the implementation of your MergeSort program by comparing it with QuickSort. To test your program you need some test data, some random lists to sort. You can create a random list of seven integer elements with the command:

```
1> RandList7 = [rand:uniform(12345) || _ <- lists:seq(1,7)].
[3632,8887,4948,9801,919,4856,7381]
```

Then you can measure the performance of your `mergesort/1` function, on a list of considerably more elements of course, by a command like the following:

```
3> timer:tc(fun () -> my_module:mergesort(RandList666) end).
{275,
 [22,43,59,71,84,92,137,181,182,216,238,246,258,267,286,297,
  305,326,328,335,339,350,360,375,383,390,391|...]}
```

Function `timer:tc/1` takes as argument an anonymous function (a `fun` containing some code to execute, it calls the fun, and returns a 2-tuple where its first element, `275` in this example, is the time it takes to execute the code of the fun and its second element is the result.

We suggest that you also compare the performance of your program with the sequential version of `qsort`. The next step is of course to parallelize your MergeSort program and measure the speedup you manage to achieve both as a function of the number of logical cores (e.g., 1..16) that can be used for schedulers in the Erlang/OTP VM (cf. the `+S` option of `erl`) and compared to the best parallel version of the QuickSort program (`pqsort4`) that we saw in the lectures. What conclusions can you draw? Just think about the answer. You do not really need to hand in anything about this part. The description of your assignment starts below.

2 Programming Assignment

The assignment involves a program that solves sudoku puzzles but you do not need to write that program! Instead, one such program is provided on the course's homepage (student portal) in a file named `sudoku.erl`. The homepage also contains a file named `sudoku_problems.txt` that contains a number of sudoku puzzles of varying difficulty, and a file named `sudoku_solutions.txt` that contains their solutions. The problem puzzles are represented as Erlang tuples; one of them is shown in Figure 1a.

```
{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]]}.
```

(a) 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}]}
```

(b) Example run of the `benchmarks` function

Figure 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 Figure 1b. Your task is to speed up their solution using parallelism.

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. 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. (If it is a machine in the lab, it suffices to mention its name.) 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 slowest to solve, 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 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`:

```
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.

3 Submission Instructions

Submit your parallelized code (your module should define and export functions: `benchmarks_parallel/0`, `solve_all_parallel/0`, and `solve_parallel/1`, which mirror the ones currently exported by `sudoku.erl`), together with a brief description of the methods you used to parallelize the solver. 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 unlimited bragging rights! Speedups which are deemed good enough by the assistant get you some of the assignment's points.

Some things to note

You probably want to pay special attention to the following:

- Make sure you use the correct formula for computing speedups.

- If your speedup seems unrealistically high, check that the machine you are testing on is not under load. More importantly, verify that your program is still producing the *correct* results; the code corrects some EUnit tests that checks that. If your program reports there is no solution to a sudoku very fast, and the sequential solver does find a solution, that is a problem.
- If you measure bad speedups then this is where you should apply the things you learnt in the course: finding sequential cutoffs, limiting parallelism where applicable, finding ways to only increase parallelism when there is not enough of it already, etc.
- Just stating that there was "no speedup" in the second part of the assignment, where the solver itself should be parallelised is *not* a sufficient answer. Even if you cannot get the performance to improve by parallelizing the solver, you can still measure a speedup (below 1) and report on that.
- For the part of the assignment that asks for a description of the methods you used to parallelize the program, a reply of the form "we parallelized the solver" is not a sufficient answer.

A checklist with what to turn in

For your convenience, here is a list of things that the assignment explicitly asks for:

Part 1: Solving the puzzles in parallel **(4 points)**

- modified source code
- output of sequential code
- output of parallel code
- the speedup that you achieved
- the characteristics of the machine this is done on
- Erlang/OTP version used
- the previous two points can be done by giving a lab machine name
- any command line options used

Part 2: Parallelising the solver itself **(4 points)**

- parallelized source code
- description of methods used
- output of parallel code
- output of sequential code
- speedup for each puzzle
- geometric mean of all speedups

The assignment has ten points in total; eight of them are shown above. The remaining two points you can get based on how good a speedup you manage to achieve in the second part of the assignment.

Have fun!