DAT280 – Lab B: “Sudoku in Erlang”

Olle Svensson (ollesv@student.chalmers.se)

Agazi Berihu (agazi@student.chalmers.se)

Attempts

Current version

Our current implementation is some sort of bruteforce version where we have replace *solve_one* with a new function *solve_all*. This function evaluates each guess in parallel and filters out a valid solution from the result.

We've also modified the parallel map in two different ways. First, it no longer cares about the order of the results, but only keeps track of how many results it expects. The order in which we receive our solutions is not important so the map might as well add the results to the list as soon as they are available. Second, it catches the *no_solution* error and excludes it from the result list.

The benchmark was done on a Macbook Pro with an Intel i5 with two physical and four logical cores.

Benchmark original:

```
{525160,
 [{wildcat,0.366},
  {diabolical,43.597},
  {vegard_hanssen,89.411},
  {challenge,6.285},
  {challenge1,346.141},
  {extreme,8.711},
  {seventeen,30.626}]}
```

Geometric mean original: 74.708

Benchmark parallelized:

```
{832707,
 [{wildcat,0.34},
  {diabolical,34.975},
  {vegard_hanssen,93.863},
  {challenge,4.214},
  {challenge1,207.041},
  {extreme,22.894},
  {seventeen,469.346}]}
```

Geometric mean parallelized: 118.377

Overall, the parallelized solution is still a lot slower since it is bruteforcing alot of unecessary work.

Future version

We have been working on another solution where the parallel map returns if it receives a valid result. Possibly this could increase performance by not searching for other solutions once one has already been found. There seems to be some race conditions still though so we do not have a working version until this deadline.

Parallelizing refine

As suggested in the PM, we started by parallelizing the refinement of rows. We did this by simply replacing the map in *refine_rows* with a parallel one. As one might have expected, the results were worse than the linear one. When analyzing the processes in percept, we can see that the lifetime for the spawned processes is really small. As another example, when solving the "diabolical" puzzle just once, we are creating 17283 processes. From this, we draw the conclusion that the task we are parallelizing is too small. We need to parallelize a bigger "chunk" of the problem.

Explanation of parallel map

In parts of the lab, we have used an existing implementation of a parallel map function taken from here. Below is an explanation of how it works.

pmap

```
pmap(F, L) ->
  S = self(),
  Pids = lists:map(fun(I) -> spawn(fun() -> pmap_f(S, F, I) end) e
  pmap_gather(Pids).
```

This is the main function called with the same arguments as a regular map, namely a function *F* to apply to each element and a list *L* with elements. To parallelize the map we want to spawn a new process for each function application so that they can be run in parallel and then let the parent process collect the results once they have been computed.

In order for the parent process to collect the results, and for the spawned processes to return their results, we need to keep track of process ID's (PID). The parent process stores its own ID in *S*. It then runs a regular map over *L*, spawning a new process for each element which executes *pmap_f*, where *S* is one of the arguments so that the spawned processes know where to send back their results. In turn, the *spawn* function returns the PID for the newly created process meaning the resulting list *Pids* of the regular map is a list of all the PID's that the parent process expects to receive results from.

Finally it calls *pmap_gather* with *Pids* as argument to collect the results.

pmag_gather

```
pmap_gather([H|T]) ->
  receive
    {H, Ret} -> [Ret|pmap_gather(T)]
  end;
pmap_gather([]) ->
  [].
```

This function is responsible for collecting the results from the processes spawned in *pmap*. It takes the first PID in its argument list and waits to receive a message from that specific process (pattern matching on *H*). Once the result *Ret* has been received it returns a list with *Ret* as head and calls itself recursively with remaining PID's to receive the next result.

pmap_f

```
pmap_f(Parent, F, I) ->
  Parent ! {self(), (catch F(I))}.
```

This is the function evaluated by each spawned process. What it does is basically to send a message back to *Parent*, which is the parent process. It attaches it's own PID by running *self* so that the parent process knows where the message is coming from. The result we want to send back in the message is retrieved by applying the function *F* to the element *I*.

It should be noted that in our context the *catch* is important. John Hughes discusses it in the Google Group here. Without it, if the spawned process gets an error (which it will do by the logic of the solver), no message would be returned. Our parallel map would be waiting for message from the process that it never receives and deadlock occurs. By adding *catch*, we are catching the error and returning it in the message instead of the result. By this the error is correctly propagated to *solve_one* where it is handled.