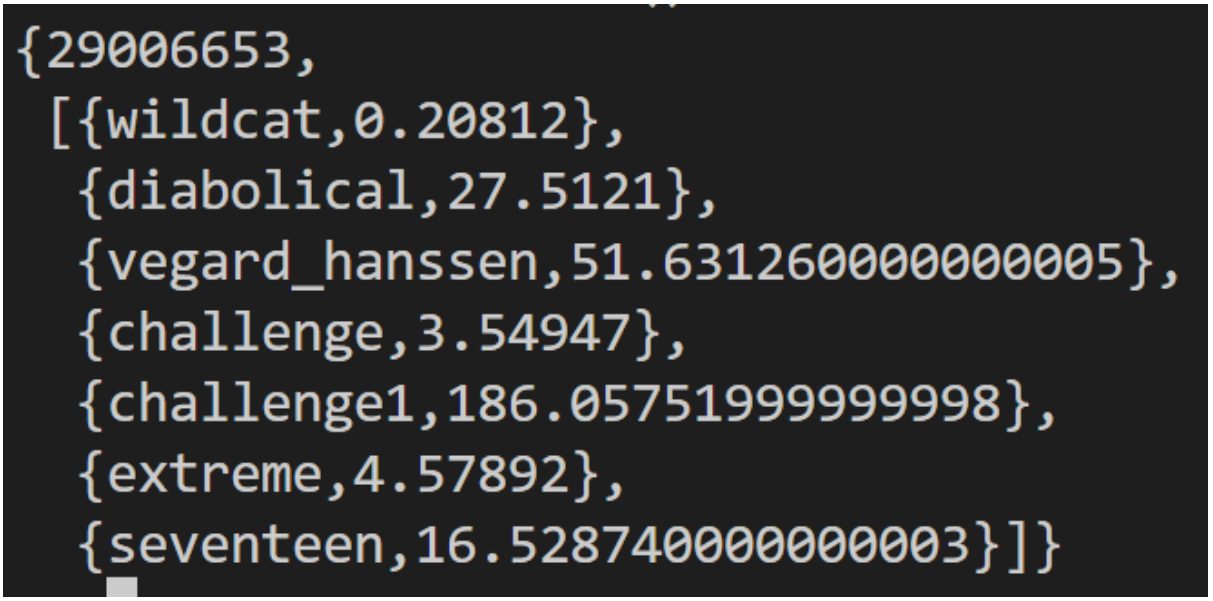
PFP Lab B: Sudoku—in parallel

Part 0: Provided sequential implementation

We fixed the EXECUTIONS value for the benchmark to 100.

It takes about 29 seconds for all the benchmarks.



Part 1: Running benchmarks in parallel

We created an erlang version of pmap (parallel map) using spawn and receive. Mapper is the mapping function and Xs are the values.

pmap_unordered simply spawns a process for each item and then receives the messages chronologically.
pmap uses pattern matching on Pid to ensure that the order is preserved.

```erlang
%% parallel map functions
%% pmap retains order, pmap_unordered doesn't preserve order (results should be ordered by end time)
%% pmap retains order because it filters the next message to receive by Pid, and Pid values are given in order by list comprehension
pmap_unordered(Mapper, Xs) ->
  Parent = self(),
  [receive Result -> Result end || _ <- [spawn(fun() -> Parent ! Mapper(X) end) || X <- Xs]].

pmap(Mapper, Xs) ->
  Parent = self(),
  [receive {Pid, Result} -> Result end || Pid <- [spawn(fun() -> Parent ! {self(), Mapper(X)} end) || X <- Xs]].
```

It is easy to rewrite the benchmarks function to use pmap_unordered / pmap, solving the sudokus in parallel.

```
benchmarks(Puzzles) ->
    pmap_unordered(fun({Name, M}) -> {Name,bm(fun()->solve(M) end)} end, Puzzles).
```

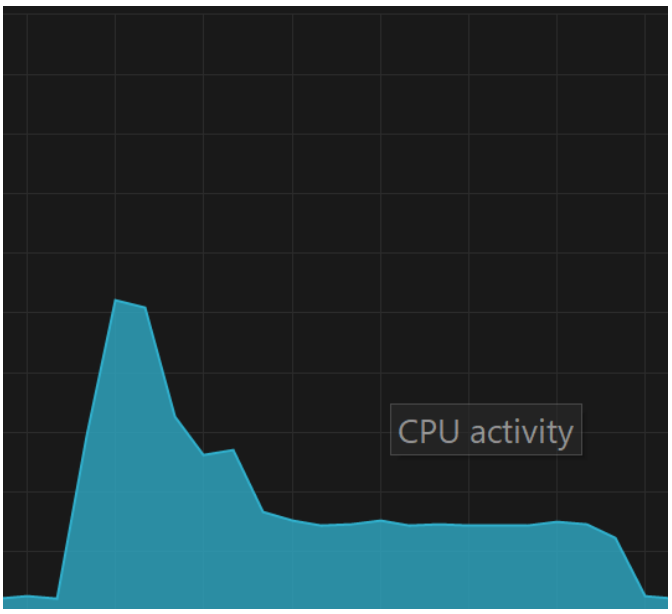
```
5> sudoku:benchmarks().
{19090910,
 [{wildcat,0.3205199999999997},
  {challenge,5.1694700000000005},
  {extreme,6.564520000000001},
  {seventeen,23.56327},
  {diabolical,28.41625},
  {vegard_hanssen,56.92367},
  {challenge1,190.90507}]}
```

The unordered version takes about 19 seconds. The resolution time of each sudoku is about the same as before.

```
benchmarks(Puzzles) ->
    pmap(fun({Name, M}) -> {Name,bm(fun()->solve(M) end)} end, Puzzles).
```

```
{22649493,
 [{wildcat,0.3174700000000003},
  {diabolical,31.44596},
  {vegard_hanssen,65.69637},
  {challenge,6.3462},
  {challenge1,226.48368},
  {extreme,7.79985},
  {seventeen,24.885060000000003}]}
```

The unordered version takes about 19 seconds. The resolution time of each sudoku is about the same as before.



Watching CPU activity, we notice that at first it is highly used as all sudokus are solved in parallel (#cores = #sudokus). Then, as soon as the easier ones finish, the cpu becomes underutilized.

Part 2: Parallelizing refine_rows

It is easy to refactor refine_rows to run in parallel with pmap (note that here we need the “ordered” version).

```
refine_rows(M) ->
    pmap(fun refine_row/1,M).
```

Since the granularity is too small, the execution didn’t finish in 5 min.

Part 3: Parallelizing solve_one

The objective is to solve all guesses in parallel, instead one at a time.

We developed pmap_fetchone, a function that works like map but returns only the first “good” result.

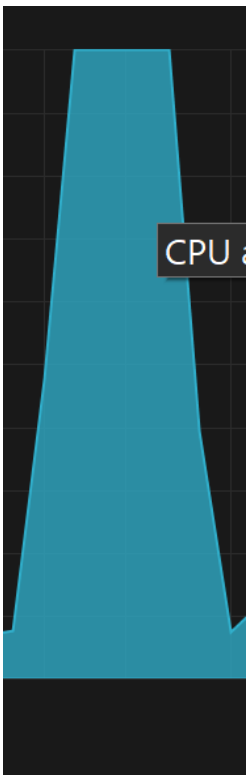
- pmap_fetchone spawns a process for each item
- fetchone receives one message at a time and returns the first non-exit
    - if all the items failed (NumWaiting = 0), no_solution is returned

psolve_one calls solve_refined in parallel and returns the first solution found or no_solution

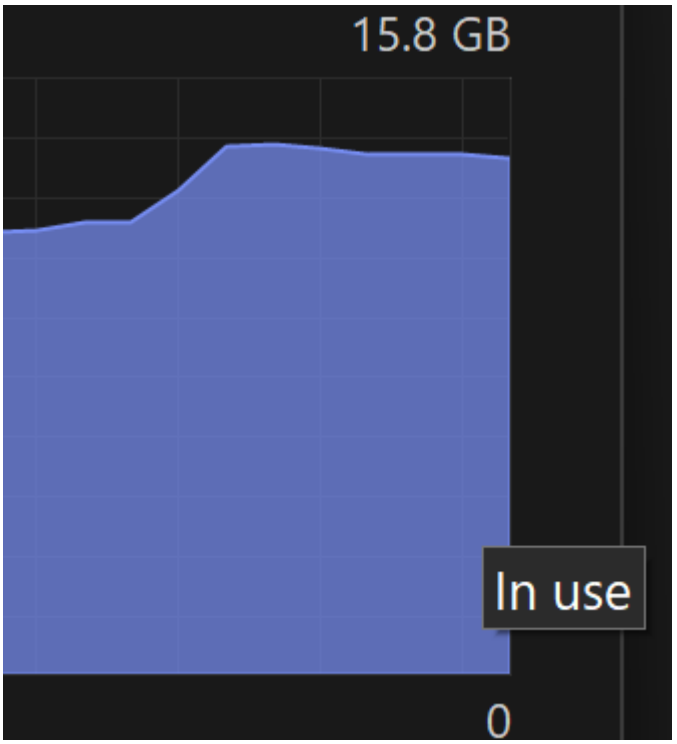
```
pmap_fetchone(Mapper, Xs)->
  Parent = self(),
  _ = [spawn(fun() -> Parent ! Mapper(X) end) || X <- Xs],
  fetchone(length(Xs)).

fetchone(0) -> exit(no_solution);
fetchone(NumWaiting) ->
    receive Result ->
        case Result of
            {'EXIT', no_solution} -> fetchone(NumWaiting - 1);
            _ -> Result
        end
    end.

psolve_one(Ms) ->
    pmap_fetchone(fun solve_refined/1, Ms).
```

Calling with EXECUTIONS=10, it runs in about 2.7 sec (so it should be about 27 sec for EXECUTIONS=100 as before).

```
{2705038,
 [{wildcat,0.1743},
  {diabolical,10.3425},
  {vegard_hanssen,23.2826},
  {challenge,2.4621},
  {challenge1,61.107800000000005},
  {extreme,8.951},
  {seventeen,164.1764}]}
```



We can see that all the puzzles but “seventeen” are a lot faster to solve (seventeen a lot slower). The cpu utilization is near 100%.



There is a problem: some memory is never freed. Trying with EXECUTIONS=100, the memory finishes and the program is killed.

We tried adding some code to kill children process when receiving a valid solution in fetchone, but the problem remains.

```
pmap_fetchone(Mapper, Xs)->
  Parent = self(),
  Pids = [spawn(fun() -> Parent ! Mapper(X) end) || X <- Xs],
  fetchone(length(Xs), Pids).

fetchone(0, Pids) -> exit(no_solution);
fetchone(NumWaiting, Pids) ->
    receive Result ->
        case Result of
            {'EXIT', no_solution} -> fetchone(NumWaiting - 1, Pids);
            _ -> [exit(Pid, kill) || Pid <- Pids], Result
        end
    end.

psolve_one(Ms) ->
    pmap_fetchone(fun solve_refined/1, Ms).
```

We will implement a full working version before the final deadline.

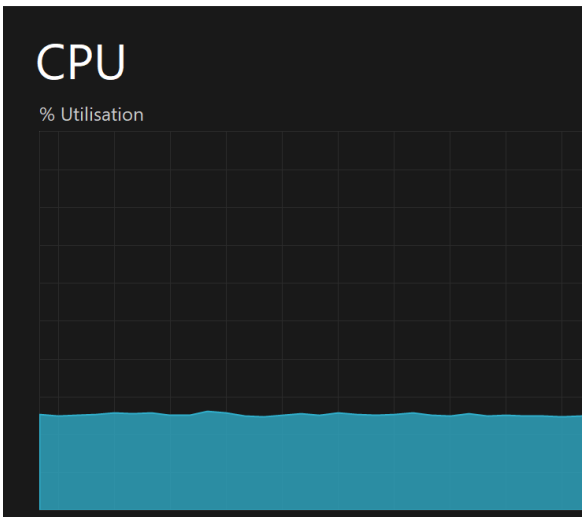
Fix 1) pmap and pmap_unordered

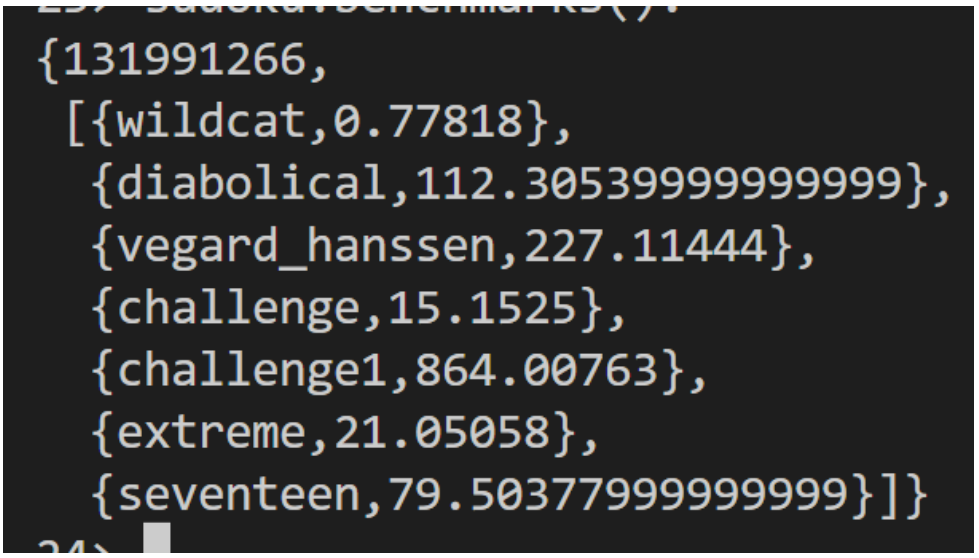
As suggested, we had a bug in pmap/pmap_unordered: if the mapping function dies (calling exit(no_solution)), pmap blocks waiting for a message forever.

We fixed the bug by adding a “catch”, which intercepts the exit messages and returns both the valid/no_solution results.

```
%% parallel map functions
%% pmap retains order, pmap_unordered doesn't preserve order (results should be ordered by end time)
%% pmap retains order because it filters the next message to receive by Pid, and Pid values are given in order by list comprehension
pmap_unordered(Mapper, Xs) ->
  Parent = self(),
  [receive Result -> Result end || _ <- [spawn(fun() -> Parent ! catch Mapper(X) end) || X <- Xs]].

pmap(Mapper, Xs) ->
  Parent = self(),
  [receive {Pid, Result} -> Result end || Pid <- [spawn(fun() -> Parent ! {self(), catch Mapper(X)} end) || X <- Xs]].
```

Now it actually works (before was just never terminating). It takes a very long time, about 120-130 sec (vs 27-29 fully sequential). The CPU usage is about 25%. Probably the reason is still the too small granularity.





fix 2)

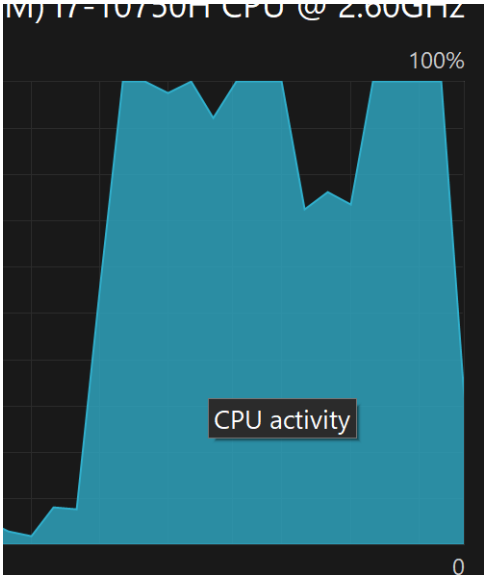
There were two problem:

- fetchone does not kill the running processes after the first solution is found → added the exit call for each Pid
- the messages were not deleted from the queue after the first solution is found → added flush function and call

```
flush() ->
    receive
        _ -> flush()
    after 0 -> ok
    end.


pmap_fetchone(Mapper, Xs)->
  Parent = self(),
  Pids = [spawn(fun() -> Parent ! Mapper(X) end) || X <- Xs],
  fetchone(length(Xs), Pids).

fetchone(0, Pids) -> exit(no_solution);
fetchone(NumWaiting, Pids) ->
    receive Result ->
        case Result of
            {'EXIT', no_solution} -> fetchone(NumWaiting - 1, Pids);
            _ -> [exit(Pid, kill) || Pid <- Pids], flush(), Result
        end
    end.
```

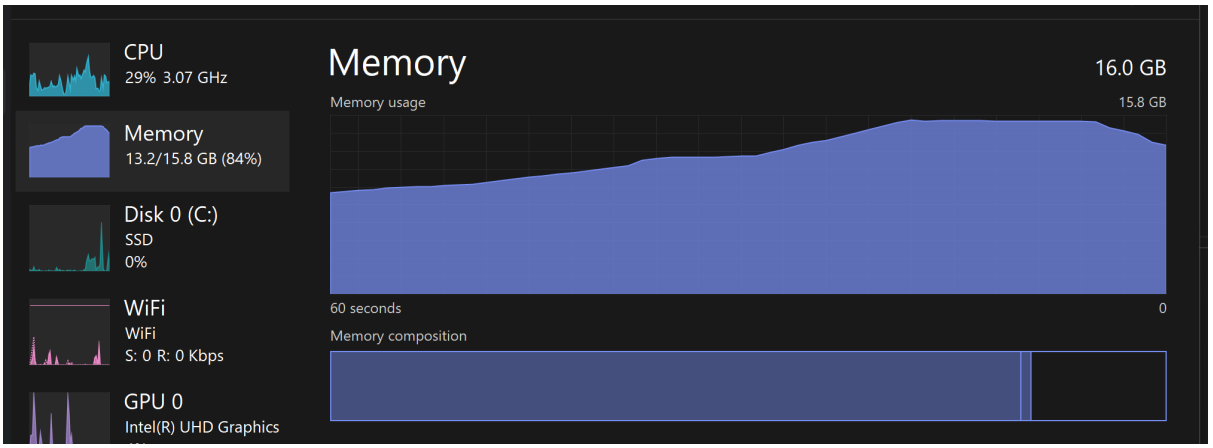
Still the program finishes memory and is killed, as before :(

We can at least notice that the cpu is used 100%.



Also putting a longer timeout, allowing the process to finish, the result is the same (but takes more time).

```
flush() ->
    receive
        _ -> flush()
    after 10 -> ok
    end.
```




```
2> sudoku:benchmarks().
Killed
```

Launching the benckmark with 10 executions, the time is 2-3.5 sec (corresponding to 20-35 sec for 100 executions).

```
{2091317,
 [{wildcat,0.21330000000000002},
  {diabolical,11.0797},
  {vegard_hanssen,21.7467},
  {challenge,2.5570999999999997},
  {challenge1,48.159},
  {extreme,6.3469},
  {seventeen,119.0218}]}
4> sudoku:benchmarks().
{3673414,
 [{wildcat,0.24609999999999999},
  {diabolical,9.466700000000001},
  {vegard_hanssen,22.9567},
  {challenge,2.6789},
  {challenge1,51.0967},
  {extreme,8.179},
  {seventeen,271.8562}]}
5> sudoku:benchmarks().
Killed
```