Lab B: Sudoku in Erlang

DAT280 - Parallel Functional Programming (Group 17)

Theodor Åstrand – theast@student.chalmers.se – 931109-9114
Rafael Mohlin – mrafael@student.chalmers.se – 931106-0017

Chalmers University of Technology

April 27, 2016

Assignment 1

For this assignment we implemented a function called par_benchmarks(). The function spawns the solve function using spawn_link recursively for each puzzle. Thereafter, the function awaits the result.

The program ran on a MacBook Pro Late 2011 with a 2,4 GHz Dual-Core Intel Core i5. In Figure 1 you can see the results of running the program in a sequential manner.

```
[2> sudoku:benchmarks().
{77666806,
 [{wildcat,0.55611},
  {diabolical,63.35106},
  {vegard_hanssen,139.10858},
  {challenge,9.69623},
  {challenge1,506.67298},
  {extreme,12.500950000000001},
  {seventeen,44.78167}]}
```

Figure 1: The benchmark; running the sudoku sequentially.

The solution depicted above can be compare to the one in Figure 2. We clearly see a difference in the total time run; the sequential benchmark had a total time of: $77666806\mu s \approx 78s$ while its parallel rivalry had a total time of: $55061426\mu s \approx 55s$. So our parallelized version of the benchmark was faster by a total of 23 seconds.

```
[2> sudoku:benchmarks().
{55061426,
 [{wildcat,3.3438600000000003},
  {diabolical,104.3547},
  {vegard_hanssen,191.17569},
  {challenge,26.066},
  {challenge1,550.60623},
  {extreme,46.62874},
  {seventeen,81.97055}]}
```

Figure 2: The benchmark; running the sudoku parallelized.

One thing that is noticeable with the result is that each individual puzzle takes longer time to solve in the parallelized version compare to the sequential. We believe this is due to that all the puzzles has to split on the computing power in the parallel solution while in the sequential solution all computational power is dedicated to one puzzle only.

In Figure 3 we can see a graph that was created using a tool called percept; on the x-axis are all the processes and on the y-axis is the total time they were active. The percept graph is created using our parallelized version of benchmark. In the graph we can see that we have 7 processes in total, one for each puzzle, and that the total active time of each processes varies depending on the difficulty of the puzzle.

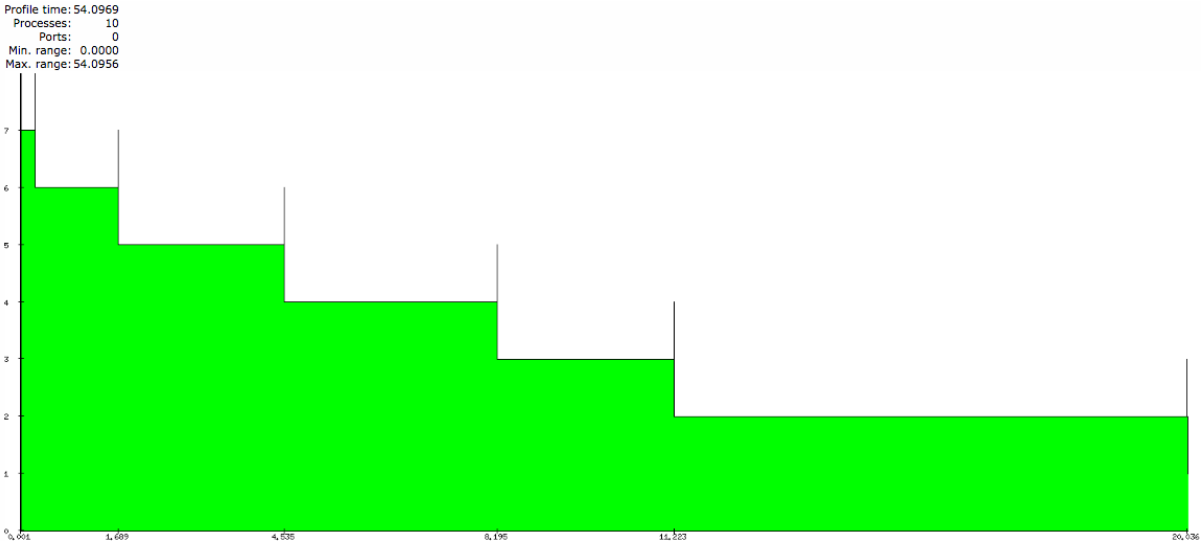

Figure 3: A graph of the processes in the benchmark.

Assignment 2

Assignment 2 is divided into three subtasks:

- 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.

Subtask 1
Naturally we started with doing the first subtask. Initially we looked at the function refine_rows(M) which contains a map function. By seeing a map function we had some flashbacks to the first lab in this course and we decided to implement a parallelized map. We spent a lot of time trying to get the parallelized map to reduce the time of the program, but the best result we got was a 2x slowdown compared to the sequential benchmark. We tried a few different concepts in order to maximize the performance of the parallelized map; implement a granularity control function so that we could handle the depth of the map, implementing "workers" so that we could kill all the processes and sub-processes when we reached an exit("no_solution"). We believe that the parallelized version of map had worse result than the sequential one because we spawned a lot of processes and that the function we applied, refine_row, did not require enough computational power to motivate a thread for each row. Instead, the parallelized map became the bottleneck in our program.

Our second approach was to create a reference list for all the rows in the sudoku, and for each element in that list we spawned a process which performed refine_row(row). Then, we received these messages in order. In Figure 4 we can see that the total time has been reduced and that the more difficult puzzles have reduced time compared to the sequential solution. We can also see that the puzzles that did not require that much time before, now requires more.

```
2> sudoku:benchmarks().
{76168743,
 [{wildcat,1.02732},
  {diabolical,62.859910000000006},
  {vegard_hanssen,139.9392},
  {challenge,10.79957},
  {challenge1,485.57203000000004},
  {extreme,14.0196},
  {seventeen,47.469370000000005}]}
```

Figure 4: The benchmark; running sudoku parallelized with par_refine_rows

Subtask 2
Parallelizing the refine method seemed like a good place to improve performance due to the job size being larger here than in refine_rows. However, the speed increase, as shown

in Figure 5, is about the same as in subtask 1. We also tested using par_refine_row and par_refine at the same time, but the benchmark results where almost identical to the results from subtask 1.

In puzzles had better performance in general with par_refine than with par_refine_rows, which is probably due to job size being larger and therefore large enough.

```
[2> sudoku:benchmarks().
 {74540360,
  [{wildcat,0.67592},
   {diabolical,66.43834},
   {vegard_hanssen,131.15971},
   {challenge,9.37913},
   {challenge1,479.84538},
   {extreme,12.64559},
   {seventeen,45.25913}]}
```

Figure 5: The benchmark; running sudoku with parallelized refine

Subtask 3
We implemented subtask 3 in a similar fashion as subtask 1. We created a reference list for each guess, and for each element in the reference list we spawned a process which performed the function refine(update_element(M, I, J, Guess)). As you can see in Figure 6 the total time increased.

```
[2> sudoku:benchmarks().
 {88952722,
  [{wildcat,1.02315},
   {diabolical,71.39263000000001},
   {vegard_hanssen,154.54886},
   {challenge,11.0085},
   {challenge1,581.41062},
   {extreme,15.13883},
   {seventeen,55.004220000000004}]}
```

Figure 6: The benchmark; running sudoku parallelized with par_guesses

Overall performance
With all of our parallellization active (except par_guesses) we arrived at our final benchmark result, which can be seen in Figure 7. The most performance was gained from solving the different sudokus in parallel, but the rest of our optimization didn't really do anything.

```
2> sudoku:benchmarks().
{55506081,
 [{wildcat,4.06751},
  {diabolical,109.56063},
  {vegard_hanssen,197.96614000000002},
  {challenge,37.298550000000006},
  {challenge1,555.05699},
  {extreme,47.69241},
  {seventeen,89.95249000000001}]}
```

Figure 7: The benchmark; running sudoku with all parallelized functions except par_guesses

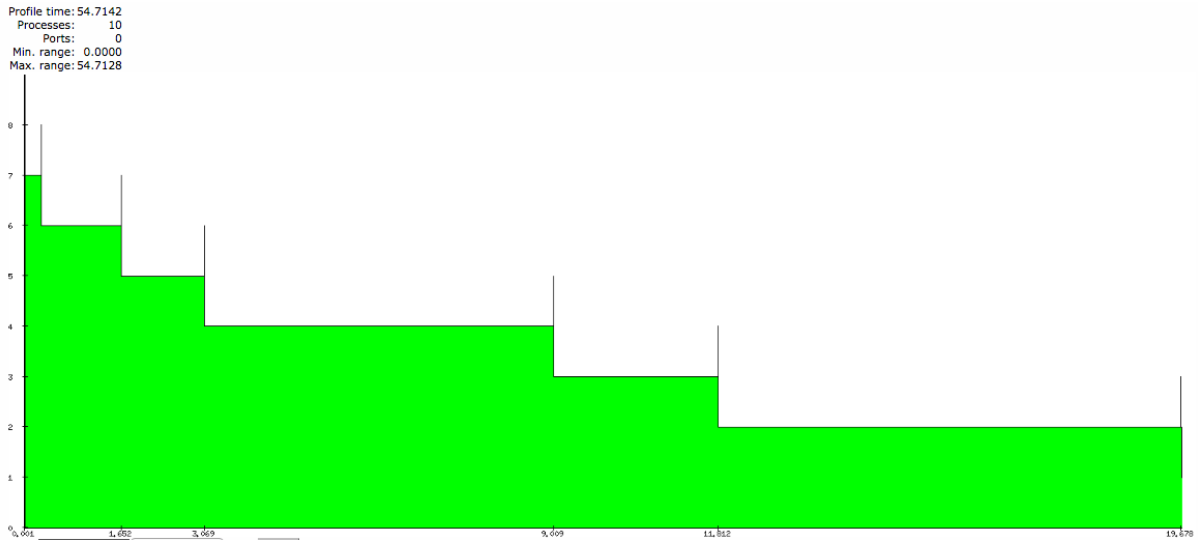

Figure 8: Percept graph using all parallelized functions except par_guesses