LabB

Adam Olsson and Sebastian Frenzel Gabrielsson

April 2020

1 Running Benchmarks in Parallel

For this part of the assignment, the machine used to run the code uses an Intel i5-8265U CPU with 4 cores on 1.60GHz and hyperthreading. Parallelizing the benchmark was done by creating a new process for each puzzle, then waiting for each of the processes to finish.

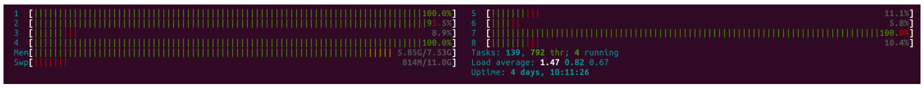
```
benchmarks_par_prime([]) -> [] ;
benchmarks_par_prime([{Name,M}]) -> [{Name, bm(fun() -> solve(M) end)}];
benchmarks_par_prime([{Name,M} | Ms]) ->
    Parent = self(),
    Ref = make_ref(),
    spawn_link(
        fun() ->
            Parent ! {Ref, bm(fun() -> solve(M) end)}
        end
    ),
    benchmarks_par_prime(Ms) ++ [{Name, receive {Ref, T} -> T end}] .
```

The results from the speed up are significant but not enormous. Because all puzzles are solved in sequence, the minimum time we can expect for the benchmarks to finish is as long it takes for the longest puzzle to finish. Additionally, because we are solving 7 puzzles on 4 physical hardware cores, 3 cores will be busy solving 2 puzzles. We could utilize this and force the most difficult puzzle to run on a single core and all other puzzles will have to share a core with another puzzle. This would probably speed up the process as well. However, right now we have no control over which puzzles are running on which core and its all up to the scheduler.

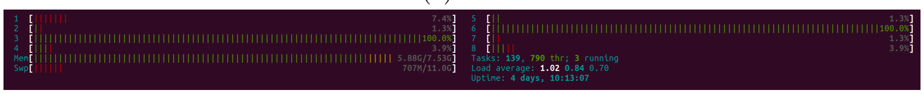
Seq	Par	Speed up
33389124	23808268	1.40

Unfortunately, we were never able to run percept to analyze the parallelization. We did manage to install it with the help from the discussion on Canvas, but whenever we tried to analyze our profile we got error saying "insert trace, unhandled". With the lack of documentation and no results in google searches

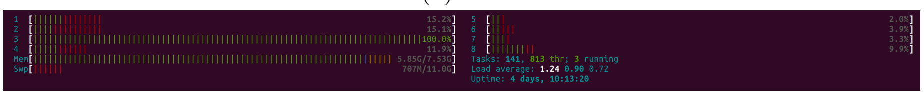
we eventually gave up. However, some insight to the parallelization be given by **htop**. Please see the screenshots from different timesteps during benchmark. In t = 0 we can see that 4/4 hardware cores are working to solve each puzzle.



(a) t=0



(b) t=1



(c) t=3

The 4 other cores are simulated cores because of hyperthreading. Once t=1, the easier puzzles has been solved which can be seen because only 2/4 cores are currently working. Here, the idle cores are waiting for new tasks which could be utilized more efficiently. But since the puzzle vary so much in length to solve and the tasks is to parallelize the benchmark, we cant get any more fine-grained control. Finally at t=3, the most difficult puzzle is the only one remaining which can be seen by the single hardware core running on 100%.

2 Parallelizing the Solver

For this part of the assignment, the machine used to run the code uses an Intel i5-8265U CPU with 4 cores on 1.60GHz and hyperthreading. The solver has been parallelized by creating new processes for each resulting matrix from guesses() returns. The recursive function *solved refined* only handles whether enough depth in the parallelization has been reached and if so, calls the sequential version *solve one*. The parallel version *solve many* maps *speculate* over the list of guesses which spawns new processes.

```
solve_refined(M) -> solve_refined(0, M) .
solve_refined(Granny, M) when Granny =< 0  ->
    case solved(M) of
            true ->
                M;
            false ->
                solve_one(guesses(M))
    end;
solve_refined(Granny, M) ->
    case solved(M) of
            true ->
```

```
                M;
            false ->
            solve_many(Granny-1, guesses(M))
    end.

solve_one([])       -> exit(no_solution);
solve_one([M])      -> solve_refined(M);
solve_one([M|Ms])   ->
    case catch solve_refined(M) of
            {'EXIT',no_solution} ->
                solve_one(Ms);
            Solution ->
                Solution
    end.

solve_many(_, [])       -> exit(no_solution);
solve_many(Granny, [M]) -> solve_refined(Granny, M);
solve_many(Granny, Ms)  ->
    Pids = [speculate(fun() ->
        catch solve_refined(Granny, X) end) || X <- Ms],
    get_value(Pids).


speculate(F) ->
    Parent = self(),
    Pid = spawn_link(fun() -> Parent ! {self(), F()} end),
    {speculating, Pid}.
```

When all processes has been created *get value* waits to receive values from the processes. Should a solution be found, all other processes are cancelled to free computational resources.

```
get_value([])           -> exit(no_solution);
get_value([Pid | Pids]) ->
    case value_of(Pid) of
        {'EXIT', no_solution} -> get_value(Pids);
        Solution ->
            cancel(Pids),
            Solution
    end .

value_of({speculating, Pid}) ->
    receive {Pid, X} ->
        X
    end.

cancel([]) -> ok;
```

```
cancel([{speculating, Pid}| Pids]) ->
    unlink(Pid),
    exit(Pid, kill),
    cancel(Pids).
```

The results from the parallelization can be seen in the table below. The best performance was reached when using a granularity value of 2. Higher values, decreased the time to solve the difficult puzzles but increased the time for the easier puzzles. It increased the time enough that the over all time also became slower.

	Seq	Par	Speed Up
Total (μs)	36983573	24820858	1.49
Wildcat (ms)	0.26331	0.33333	0.78
Diabolical (ms)	28.08293	23.51574	1.19
Vegard Hanssen (ms)	60.43089	43.32144	1.39
Challenge (ms)	4.21018	3.97426	1.06
Challenge1 (ms)	248.55875	125.71863	1.97
Extreme (ms)	5.84084	9.4469099	0.62
Seventeen (ms)	22.44857	23.590130	0.95
Geometric Mean			1.53

Interestingly enough, this method performs worse on some sudokus. We believe that this is because many processes are created whose search space are shallow. These processes have finished their search within a short time, resulting in that the overhead of creating these processes are more expensive than running them in sequence. Moreover, on the Challenge1 sudoku our methods performs the best. This is most likely because the search space for the spawned processes are deep and we can utilize the parallelization.

Another method we tried to speed up the solver was to use throttling to the amount of hardware threads available. Unfortunately, did not manage to get it working properly and only 2 out of 8 hardware threads was actually used.