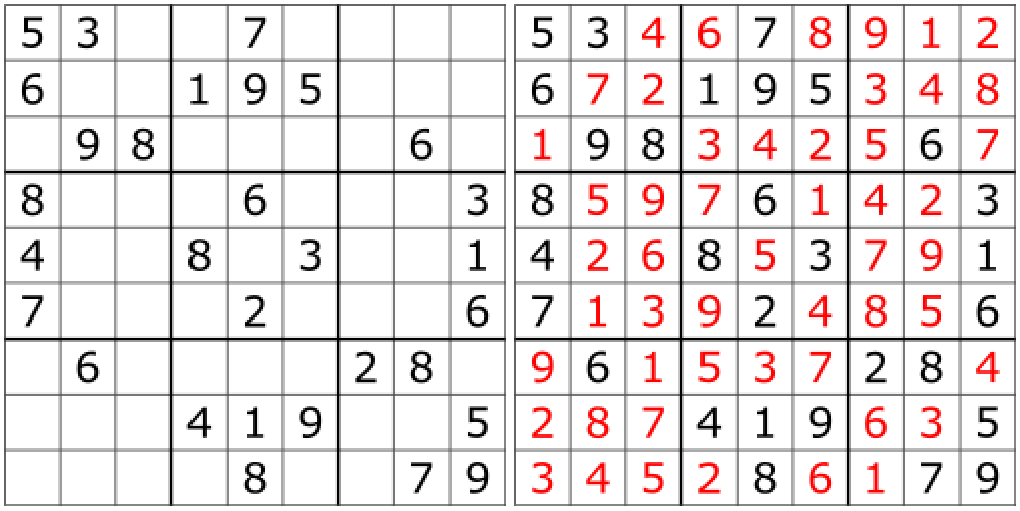
19 Fall Parallel Functional Programming Project Report

Parallel Sudoku Solver in Haskell

Jiaheng He(jh4003)

Project Background

Sudoku is a logic based, number placement puzzle. The objective of this game is to fill a 9*9 grid with digits that satisfy some rules, like no duplicate elements in a row/column/box.



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

It's easy to understand as it looks. Sudoku can be modeled as a constraint satisfaction problem, and also can be solved with naive backtracking, even possible with heuristic search if you like. There are many ways to solve this problem, but their efficiency may vary a lot.

Project Goal

I chose to take this course because I didn't have any functional programming experience. And I did some multi-thread programming and performance related projects. As far as I know, some object oriented programming language like Golang, C++, Python has some functional programming paradigm embedded in. And sometimes it's hard for me to follow a fixed programming paradigm. Because some code looks like functional and some looks like object oriented, and they are in the same project. And some people say that it's easy to use FP to solve parallel computing problems.

So I'd like to take this chance to learn:

1. How to write parallel and concurrency code(example question: I've read some posts about add -threaded when compile may make the program faster, but there isn't any code start a new thread)
2. Pros and Cons of FP(efficiency, maybe easy to code for parallel and concurrency?)
3. When to use FP? What kind of projects is more suitable for FP?

Instead of choosing a hard problem to solve, I'd like to take this simple problem and try to find out the answers of these three problems. So that I can focus more on the language rather than the problem.

Sudoku solver algorithms

1. Dancing Links

   Dancing links is a data structure invented by Donald Kruth for his project Algorithm X, which is a backtracking solution for the exact cover problems. And N Queens problem and Sudoku are both exact cover problems. On Hackage I found a package called "exact-cover" aiming to solve exact cover problem. But learning to use a package wouldn't help me know more about haskell efficiency, and I better know how to use the language first before I implement an algorithm with it. Otherwise, I can only implement the algorithm as the pseudo-code describes and it's hard for me to optimize it and make full use of syntax sugar from Haskell.

2. Naive backtracking

   To me, this is still a good point to start with. I wouldn't say I can write efficient backtracking code in Haskell. So it's a good point for me to start and improve my code. Dancing links can count as a backtracking algorithm, and it's also a search algorithm. How you make the next guess and prune unnecessary branch can largely affect the running time of your code.

3. Constraint Satisfaction Problem

   CSP is also a backtracking problem, on finite domains, it's typically solved using a form of search(backtracking, constraint propagation). In Sudoku, you may have multiple options for a black cell, and your choice for this cell will definitely affect constraints set of other cells. Constraint propagation is a method that prune unnecessary branches and reduce search time based on your choice on current cell.

Code

I managed the project with the help of stack. A simple usage of the code is in "README.md", and source code is in "app/Main.hs" and "src/Sudoku.hs", test cases are in "test/Spec.hs", and I ran multi-core test with input from "easy.txt", "hard.txt".

Main.hs

```
module Main where

import Control.Applicative
import Control.Monad
import System.Environment
import System.IO

import Sudoku (solve)

-- Solve Sudoku puzzles from a file of Sudoku strings
main = do
  [f] <- getArgs
  lines <$> readFile f >>= mapM_ (mapM_ putStrLn . solve)
```

Sudoku.hs

```
module Sudoku(solve) where

import Data.List
import Data.Char

type Cell = Maybe Int
type Row a = [a]
type Grid a = [Row a]
type Options = [Cell]

boxes :: Grid a -> Grid a
boxes = deser . (map transpose) . ser
    where
    ser = (front 3) . map (front 3)
    deser = map concat . concat
    front _ [] = []
    front n xs = take n xs : front n (drop n xs)
```

```
filter_elem :: [Options] -> [Options]
filter_elem ops = [op `minus` single | op <- ops ]
   where
   single = concat (filter (\l -> length l == 1) ops)
   op1 `minus` op2 = if length op1 == 1 then op1 else op1\\op2

-- filter duplicate pairs
filter_pair :: [Options] -> [Options]
filter_pair x = [if (isInfixOf dup_pair xs) && (dup_pair /= xs) then xs\\dup_pair else xs | xs <- x]
   where
   dup_pair = concat $ getDups $ filter (\l -> length l == 2) x
   getDups t = t \\ nub t

-- pruneWith prune by (func) rule
prune :: Grid Options -> Grid Options
prune = (pruneWith filter_pair) . (pruneWith filter_elem)
   where
   pruneWith func = pruneBy id . pruneBy transpose . pruneBy boxes
      where
      pruneBy f = f . map func . f

allOptions :: Grid [a] -> [Grid a]
allOptions g = cardProd $ map cardProd g
   where
   cardProd [] = [[]]
   cardProd (x:xs) = [y:ys | y <- x, ys <- cardProd xs]

-- start with the cell have fewest choices
try :: Grid Options -> [Grid Options]
try ops  = [rows_t ++ [row_t ++ [c] : row_aft] ++ rows_aft | c <- cs]
   where
   (rows_t, row:rows_aft) = break (any fit) ops
   (row_t, cs:row_aft) = break fit row
   fit t = (length t == len)
   len = minimum . filter(>1) . concat $ map (map length) ops

search :: Grid Options -> [Grid Cell]
search ops
   | any (any null) ops || not (success ops) = []
   | all (all (\l -> length l == 1)) ops = allOptions ops
   | otherwise = [s | ops' <- try ops, s <- search (arriveAt prune ops')]
   where
      arriveAt f x
```

```
          | x == f x = x
          | otherwise = arriveAt f (f x)
        success t = all good (t) && all good (transpose t) && all good (boxes t)
        good = noDup . concat . filter (\l -> length l == 1)
        noDup l = nub l == l

-- convert grid cell to sudoku string
deFormat :: Grid Cell -> String
deFormat cl = toStr $ concat cl
    where
    toStr [] = ""
    toStr (Nothing:xs) = "." ++ toStr xs
    toStr (Just a:xs) = [intToDigit a] ++ toStr xs

-- convert sudoku string to grid cell
format :: String -> Grid Cell
format xs
    | length xs == 0 = [[]]
    | otherwise = (takeS 9 xs) : (format (drop 9 xs))
    where
        takeS :: Int -> String -> Row Cell
        takeS 0 _ = []
        takeS _ [] = []
        takeS _ [x] = convert x
        takeS n (x:xt) = convert x ++ takeS (n-1) xt
        convert x = if x == '.' then [Nothing] else [Just (digitToInt x)]

solve :: String -> [String]
solve str = mtxToStr matrix
    where
        matrix = search $ prune $ options $ format str
        mtxToStr mtx = [deFormat m | m <- mtx]
        options = map (map cur)
        cur v = if v == Nothing then [Just i | i <- [1..9]] else [v]
```

Spec.hs(test file)

```
import Sudoku

eg1 :: String
eg1 = "3.6.7...........518.........1.4.5...7.....6.....2......2.....4.....8.3.....5....."
eg2 :: String
eg2 = "1.....3.8.7.4..............2.3.1...........958.........5.6...7.....8.2...4......."
```

```
eg3 :: String
eg3 = ".237....68...6.59.9.....7......4.97.3.7.96..2.........5..47.........2....8......."

main :: IO ()
main = do
   runTest eg1
   runTest eg2
   runTest eg3

runTest :: String ->  IO ()
runTest eg = do
   putStrLn $ "running test"
   putStrLn $ "case:   " ++ eg
   putStrLn $ "answer: " ++ $ concat $ solve eg
```

Explanation of my solution

A sudoku problem is described as a 81 characters string. And the "solve" function takes in a sudoku string and return a list of solutions in the format of strings. For a single Sudoku problem, I start with a brute force backtracking solution. At first, I take in the sudoku string and parse it into a 9*9 Maybe grid. Cell will no number filled in will be filled in with "Nothing". Then I filled in all options(from 1 to 9) into "Nothing" cells, then start brute force searching until every cell only has one option left. This is the 1st version of my solution and it's extremely slow. Then I start to prune some unnecessary branches to save some search time, which is called constraint propagation. If you make a choice on current cell, then choices on cells on the same row/column/box will be affected. So I delete invalid options before next search to save some time. And when I start searching, I start will the cell has the fewest options left, which will also make the search faster.

Test

Test environment:
OS: Ubuntu 18.04.1
Mem: 16GB
CPUs: 8 vCPUs

1. Comparing with online solution for a single problem

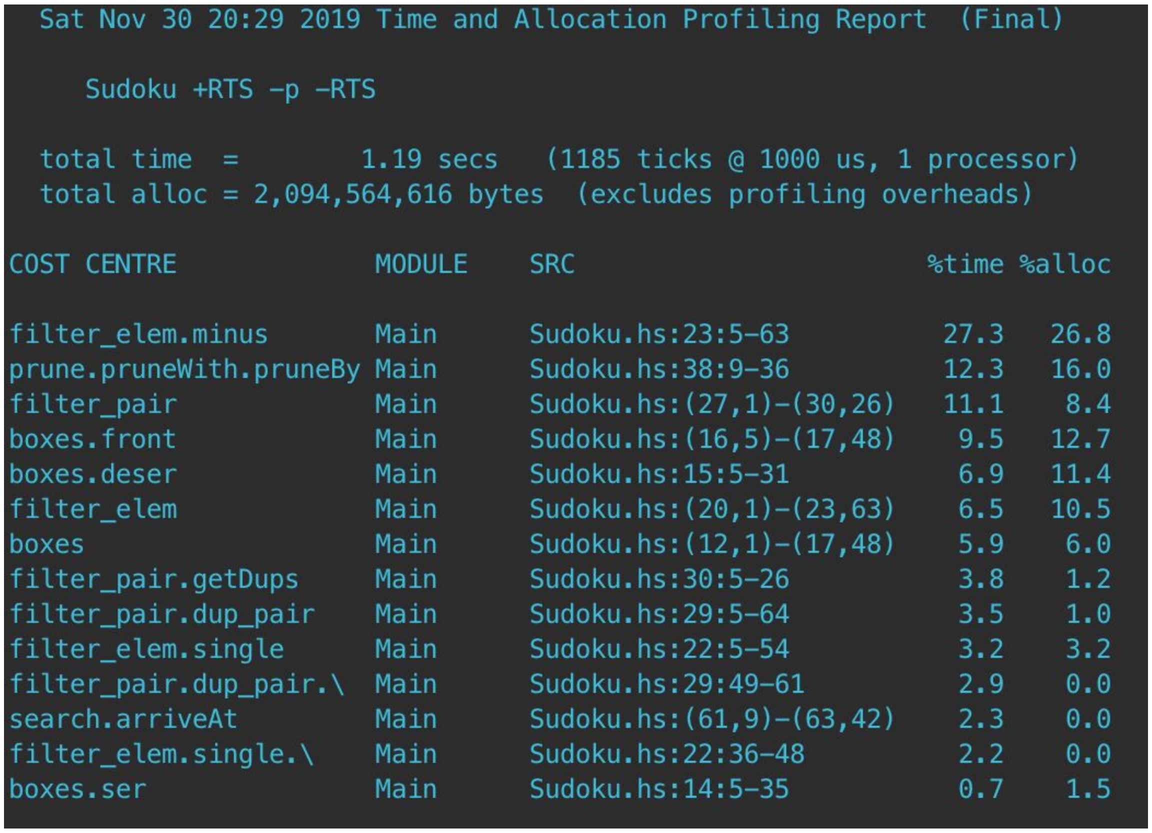
   A Constraint Propagation solution by Peter Norvig:
   solving same Sudoku problem:

".......2143.......6........2.15..........637...........68...4.....23........7...."

	running time - till the solution is printed
Peter Norvig's	0.025s
mine	0.938s

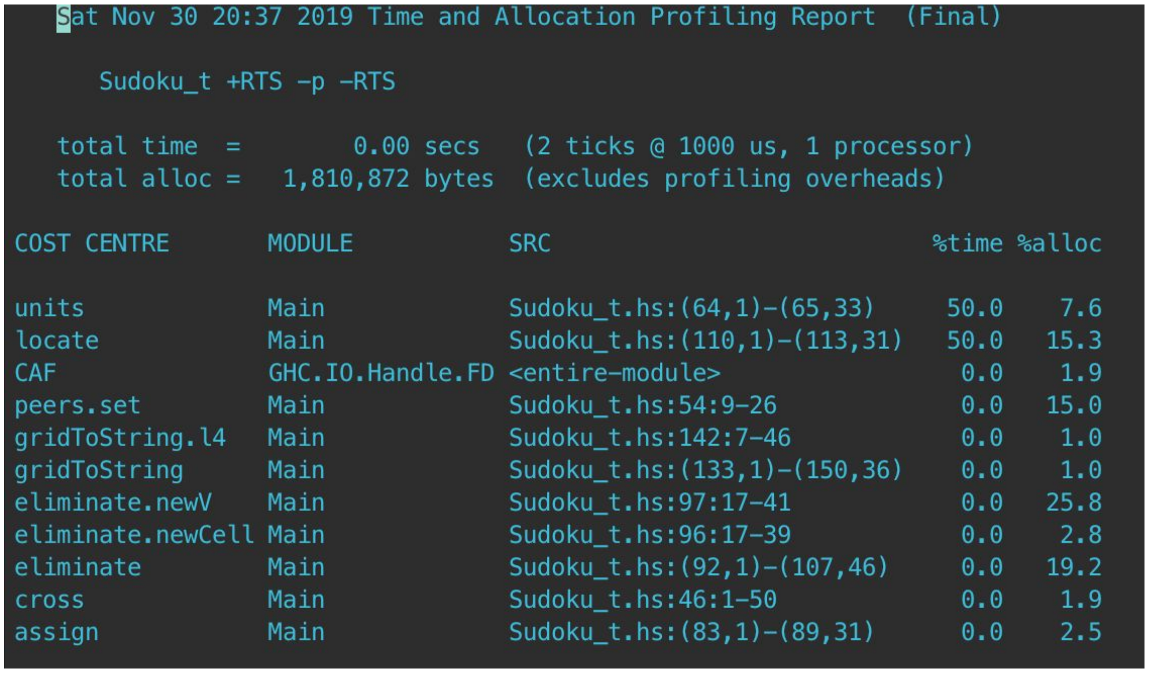
I tried a lot to optimize my logic and prune more branches before searching, but my solution is still much more slower than Peter Norvig's. It seems to me that the complexity of Haskell is not that observable and don't similar to the algorithm logic. And this is different in object oriented languages. In OO languages and other procedure oriented languages, it's easier to analyze running time. I searched online about this problem and one answer is reasonable to me. It says that OO and PO languages are based on turing machine, which is easier to analyze time complexity. So most algorithms we learned in "Introduction to Algorithms" so far are based on this, and it's easier to learn. But Haskell and other languages are based on lambda calculus. There's a paper talked about this, and it's hard and meaningless to apply Big O notation on functional languages. So this is the reason why two programs' running time may differ a lot even if they follow the same pseudo code. But how can we write efficient code? Do we have to run profiling everytime to check the bottleneck? Or maybe functional programming languages shouldn't be used at efficiency critical situation? There's a book called "Pure functional data structure" and from that I know the data structures behind functional languages is totally different from what I used before. Simply applying my experience from that is definitely a wrong decision.

Profiling for my final solution:



```
  Sat Nov 30 20:29 2019 Time and Allocation Profiling Report  (Final)

     Sudoku +RTS -p -RTS

  total time  =        1.19 secs   (1185 ticks @ 1000 us, 1 processor)
  total alloc = 2,094,564,616 bytes  (excludes profiling overheads)

COST CENTRE             MODULE    SRC                          %time %alloc

filter_elem.minus       Main      Sudoku.hs:23:5-63             27.3   26.8
prune.pruneWith.pruneBy Main      Sudoku.hs:38:9-36             12.3   16.0
filter_pair             Main      Sudoku.hs:(27,1)-(30,26)      11.1    8.4
boxes.front             Main      Sudoku.hs:(16,5)-(17,48)       9.5   12.7
boxes.deser             Main      Sudoku.hs:15:5-31              6.9   11.4
filter_elem             Main      Sudoku.hs:(20,1)-(23,63)       6.5   10.5
boxes                   Main      Sudoku.hs:(12,1)-(17,48)       5.9    6.0
filter_pair.getDups     Main      Sudoku.hs:30:5-26              3.8    1.2
filter_pair.dup_pair    Main      Sudoku.hs:29:5-64              3.5    1.0
filter_elem.single      Main      Sudoku.hs:22:5-54              3.2    3.2
filter_pair.dup_pair.\  Main      Sudoku.hs:29:49-61             2.9    0.0
search.arriveAt         Main      Sudoku.hs:(61,9)-(63,42)       2.3    0.0
filter_elem.single.\    Main      Sudoku.hs:22:36-48             2.2    0.0
boxes.ser               Main      Sudoku.hs:14:5-35              0.7    1.5
```

The "minus" function is very slow, but even if I can improve its speed as other functions like "boxes", it still can't catch up with Peter Norvig's solution.
And this is the profiling for his solution:



```
   Sat Nov 30 20:37 2019 Time and Allocation Profiling Report  (Final)

      Sudoku_t +RTS -p -RTS

   total time  =        0.00 secs   (2 ticks @ 1000 us, 1 processor)
   total alloc =   1,810,872 bytes  (excludes profiling overheads)

COST CENTRE       MODULE           SRC                           %time %alloc

units             Main             Sudoku_t.hs:(64,1)-(65,33)     50.0    7.6
locate            Main             Sudoku_t.hs:(110,1)-(113,31)   50.0   15.3
CAF               GHC.IO.Handle.FD <entire-module>                 0.0    1.9
peers.set         Main             Sudoku_t.hs:54:9-26             0.0   15.0
gridToString.l4   Main             Sudoku_t.hs:142:7-46            0.0    1.0
gridToString      Main             Sudoku_t.hs:(133,1)-(150,36)    0.0    1.0
eliminate.newV    Main             Sudoku_t.hs:97:17-41            0.0   25.8
eliminate.newCell Main             Sudoku_t.hs:96:17-39            0.0    2.8
eliminate         Main             Sudoku_t.hs:(92,1)-(107,46)     0.0   19.2
cross             Main             Sudoku_t.hs:46:1-50             0.0    1.9
assign            Main             Sudoku_t.hs:(83,1)-(89,31)      0.0    2.5
```

And his solution is not the fastest among all Sudoku solvers. How to write efficient and elegant Haskell code is still an open question to me. :)

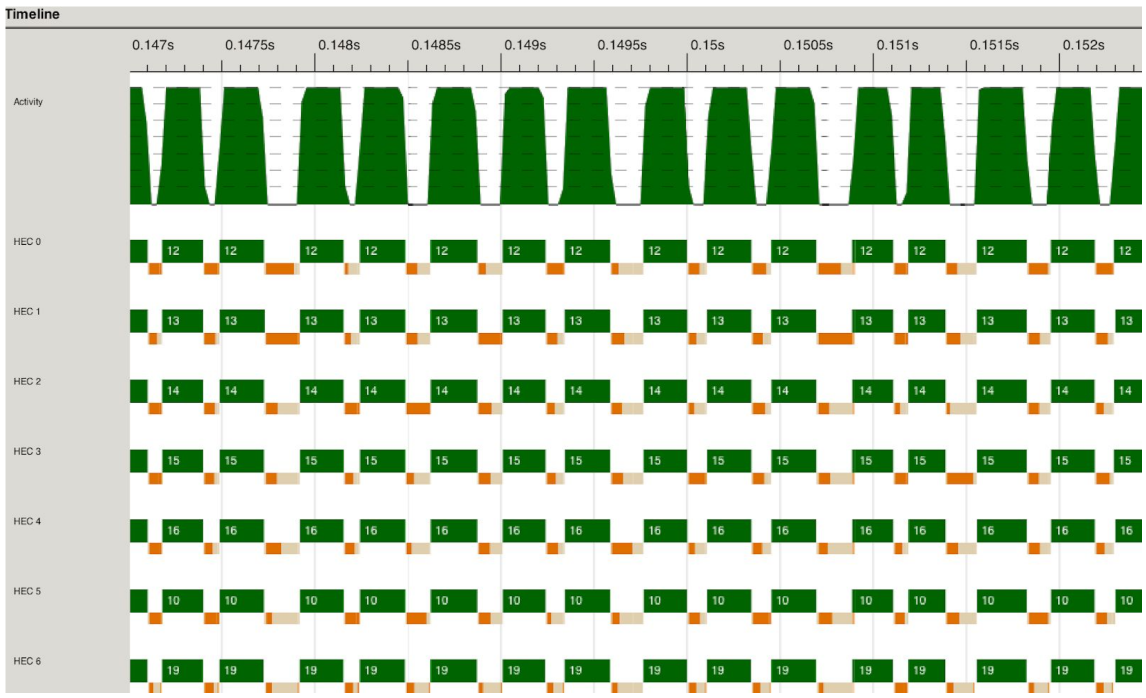
Then I dig deeper about Peter Norvig's solution. It turns out his original solution is in Python and Emmanuel ported his thoughts to Haskell. And here is the email thread about how did Emmanuel and some gurus improve his code. I read all those emails, though they made the code very fast at last, they still can't draw a conclusion why it ran fast like that. Some said using array is faster than list in this problem, some said a better algorithm is a better optimization. My conclusion is, I should know better about data structures/packages commonly used in Haskell, and learn more about functional programming paradigm and functional thoughts. And I gradually agree with Yitz said in the email thread: "But I personally find that for my own purposes, pure, simple, clear Haskell is almost always more than fast enough. And it saves truckloads of debugging time."

2. Running parallely

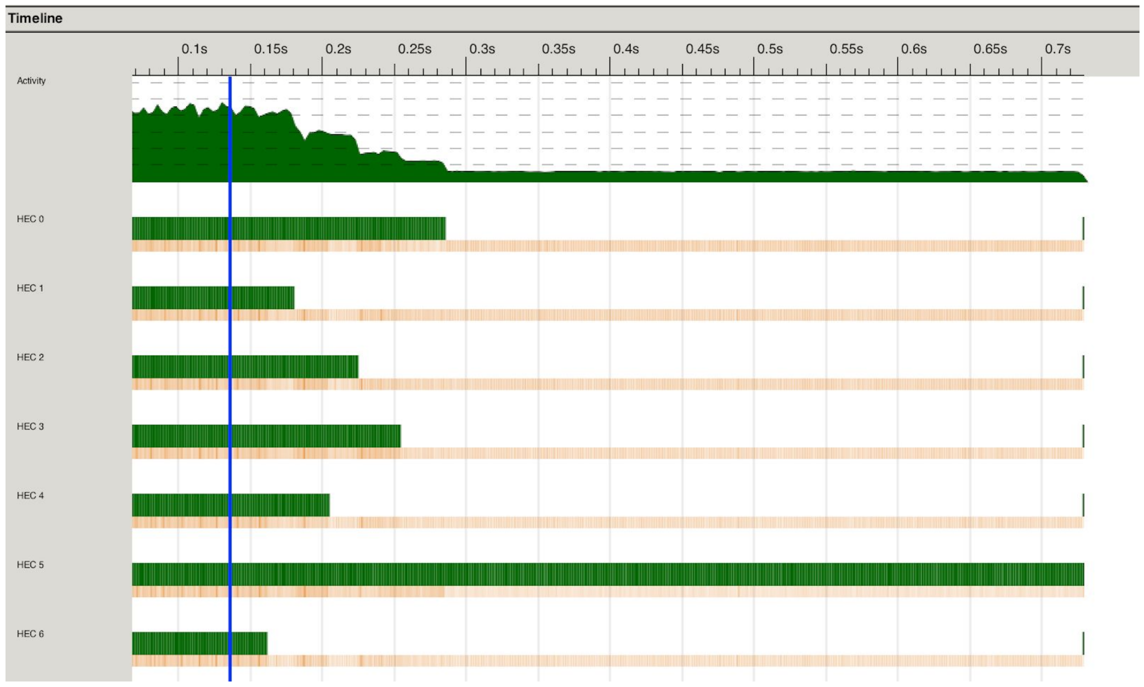
Before I start this project, I don't know why Haskell divided into parallel and concurrent programming two parts. But after I read this notes, which is written by Simon Marlow(author of Parallel and Concurrent Programming in Haskell), I think I gradually understand why it is designed like this. Parallel is typically used when you want to run a program on multiple core. Although in some occasions in OO languages, we start multiple processes to run faster instead of using multiple threads. I think the idea is similar but the mechanism is different. This kind of parallel requires no data exchanging in sub-problems. And in Haskell, spark pool is implemented as a ring buffer. Though I'm not sure will there be "fork" system call happens or not when you use spark in Haskell on multi-cores. But more memory level operation and less fork is always good for efficiency. And concurrency is different, it supports thread communication but at the cost of efficiency. So for this kind of problem, like solve many Sudoku/Kenken problems, and web crawler, I would like to use parallel. But parallel granularity is still a hard problem.

Because my solution is too slow, 1000 problems will take forever, so I tested with much less problems in the following tests.

50 easy problems running with 7 cores:

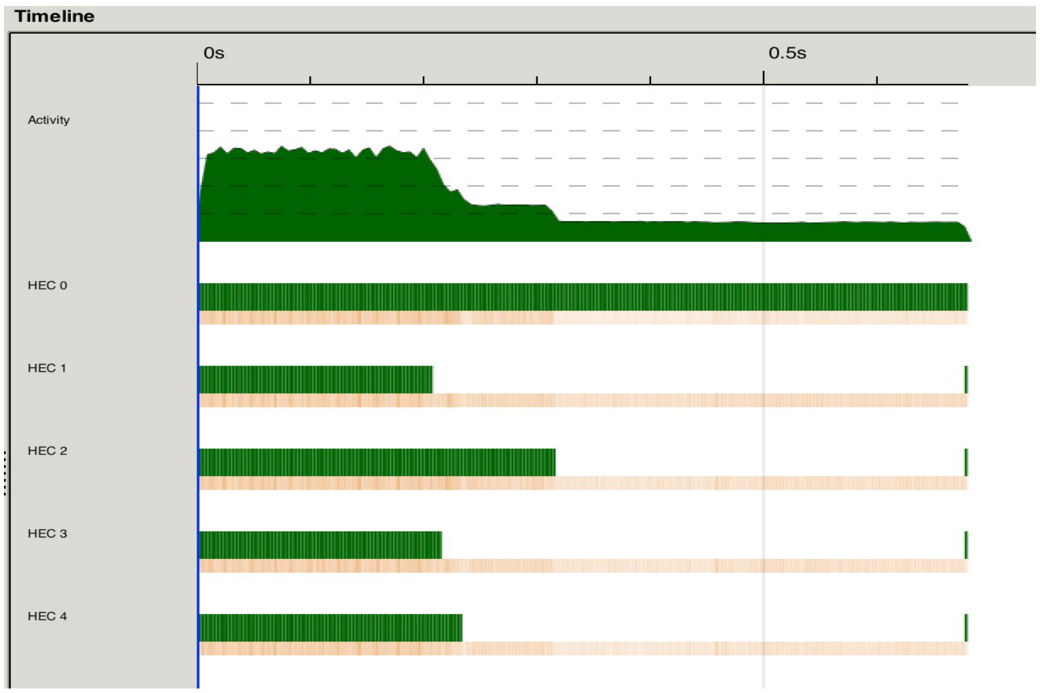


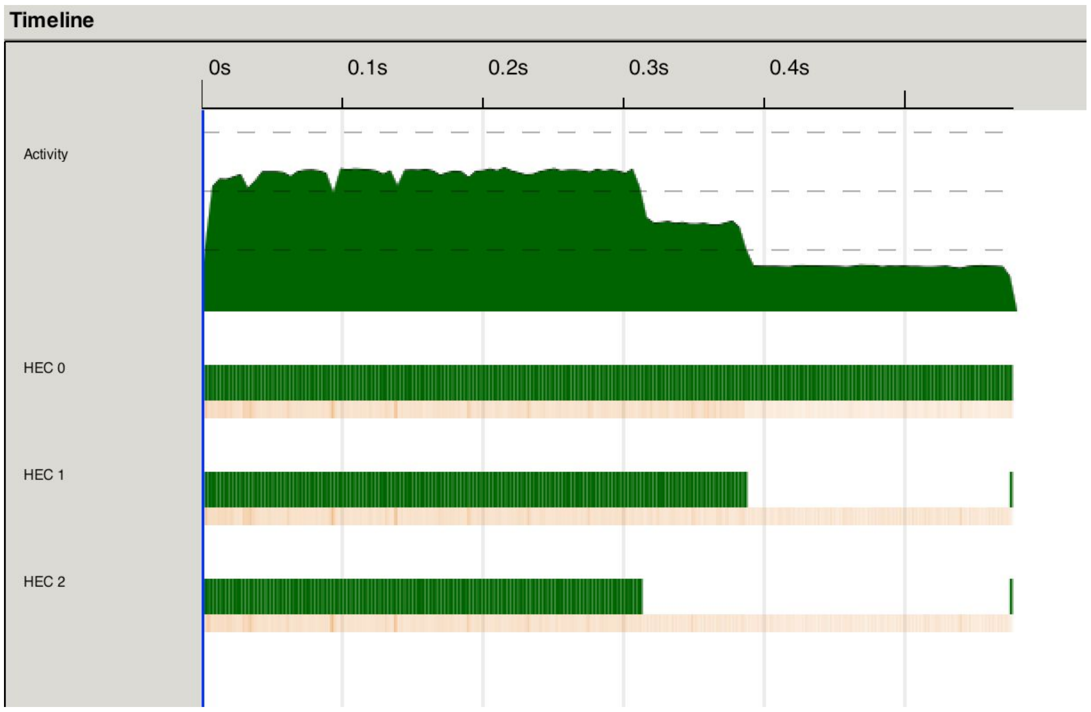
From this we can say that tasks are evenly dispatched.



And from this graph, we can see that HEC5 ran for quite a long time, for job 10. This search takes too long, so that running time on different core differs a lot.

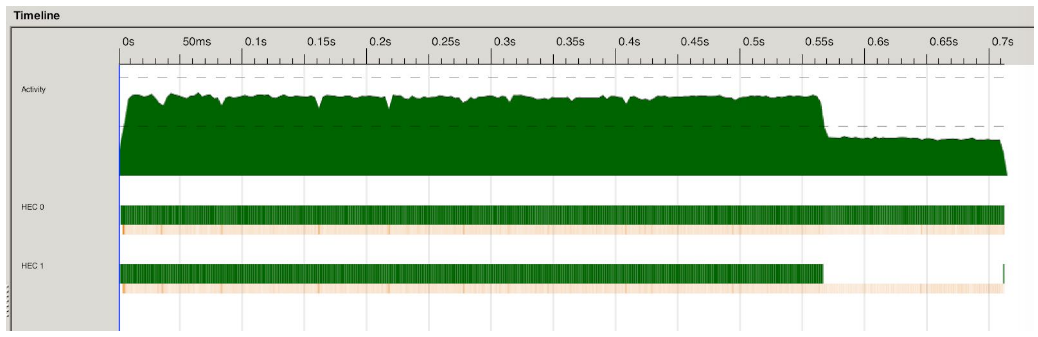
For 5 cores and 3 cores are similar.





But 3 cores is the fastest among these 3 tests.

For 2 cores:



So running with 3 cores is (maybe) the fastest among all my test. I think for some Sudoku problems, search with my algorithm is very slow. So how did it dispatch tasks will affect total running time a lot.

Acknowledgement

I'd like to thank Prof. Edwards and TAs for their teaching and homework design. I have to say homework are very interesting. This course shows me a brand new programming paradigm, and gives a good start on functional programming languages. I have to admit that I don't have enough to finish the concurrent programming part as I discussed in the project proposal. I need to do a surgery 2 days later and have to be lying in bed for 2 weeks or more. So I just tried to finish this project as fast as I can. This is the last course I have at Columbia. I'm so glad to meet all the amazing professors and classmates, and it's a memorable journey. I did my best on this project and got what I wanted to learn, though if I have enough time I can definitely do more. :) It's a pity that I can't attend last several classes. Thank you for reading this report and wish you all the best!