Get Statistics of 8-puzzle Problem Using Parallel MapReduce

Name: Yimin Hu
UNI: yh3052

1. Parallel MapReduce in Haskell

The basic idea of MapReduce is splitting the work and assigning them to different workers. Each worker finishes its own task. The result will be merged at the end of the process. In this project, I implemented a simple version of generalized MapReduce which runs parallel on multiple cores. Note that it's not identical with the real world MapReduce, which a single mapper normally is just a single machine.

MapReduce consists of these main stages: map, shuffle, reduce. It sometimes has parse stage at the beginning and merge stage at the end. Base on that, I implemented a simple MapReduce like this:

```
mapReduce :: (NFData a, NFData b, NFData c, NFData d) =>
    (a -> b) -- mapper
    -> ([b] -> [c]) -- shuffle
    -> (c -> d)     -- reducer
    -> [a] -- state
    -> [d] -- result

mapReduce mapFunc shuffleFunc reduceFunc input = mapResult `pseq` reduceResult
  where mapResult = parMap rdeepseq mapFunc input
        shuffleResult = shuffleFunc mapResult
        reduceResult = parMap rdeepseq reduceFunc shuffleResult
```

The idea is that map and reduce stage is compatible with parallel computing, while shuffle usually requires the full data to run it sequentially. By using parMap and rdeepseq, map and reduce stage will use dynamic partitioning strategy to utilize multiple cores. For specific applications, they only need to plug in their mapper, shuffler and reducer to get the result.

2. A practical MapReduce problem and its performance

One of the most frequently used MapReduce application is analyzing the http logs. I implemented a program to get the visit frequency of IP to a server through running a MapReduce on its server log.

As each line starts with an IP address, this application is straightforward. Parse each line and map IP to (IP, 1). After shuffle those key value pair and group them by key. Finally sum over the value list to get the frequency. The nature of this application is very similar to a word count program.

The question is whether parallelization helps speed up this program. I wrote both sequential and parallel version to test. Here's the result of -N1 and -N2:

```
INIT    time    0.001s  (  0.004s elapsed)
MUT     time    0.309s  (  0.337s elapsed)
GC      time    1.307s  (  1.467s elapsed)
EXIT    time    0.000s  (  0.008s elapsed)
Total   time    1.617s  (  1.816s elapsed)

Alloc rate    5,102,245,529 bytes per MUT second

Productivity  19.1% of total user, 18.5% of total elapsed
```

Fig 1. ipFreq -N1

```
INIT    time    0.001s  (  0.004s elapsed)
MUT     time    0.456s  (  0.461s elapsed)
GC      time    1.519s  (  0.858s elapsed)
EXIT    time    0.000s  (  0.007s elapsed)
Total   time    1.976s  (  1.330s elapsed)

Alloc rate    3,457,455,669 bytes per MUT second

Productivity  23.1% of total user, 34.7% of total elapsed
```

Fig 2. ipFreq -N2

As we can see, although in the term of elapsed time for -N2 is indeed shorter comparing to -N1, this doesn't mean there's a speedup in parallel. A sequential version is much faster with less than 0.9s to accomplish this task. The stats show that GC is dominant in parallel version and happens much more in -N1 situation, which is very hard to overcome while the application itself needs to read in a huge file whereas the computation itself is actually very cheap. Threadscope diagrams shows how GC affect the result, even if the work is indeed distributed over cores:

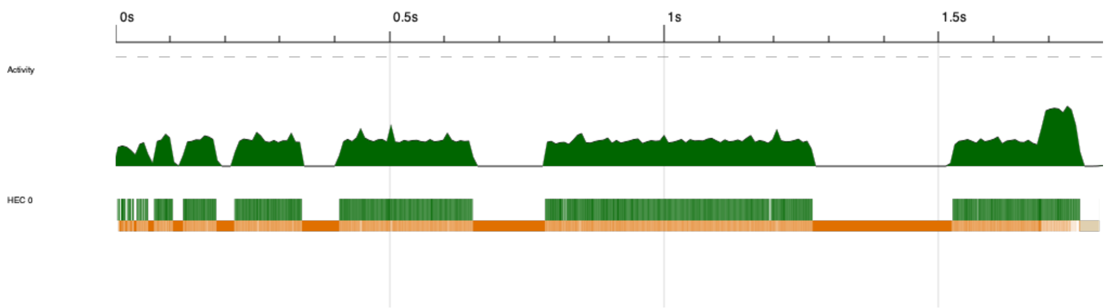

Fig 3. ipFreq -N1 threadscope

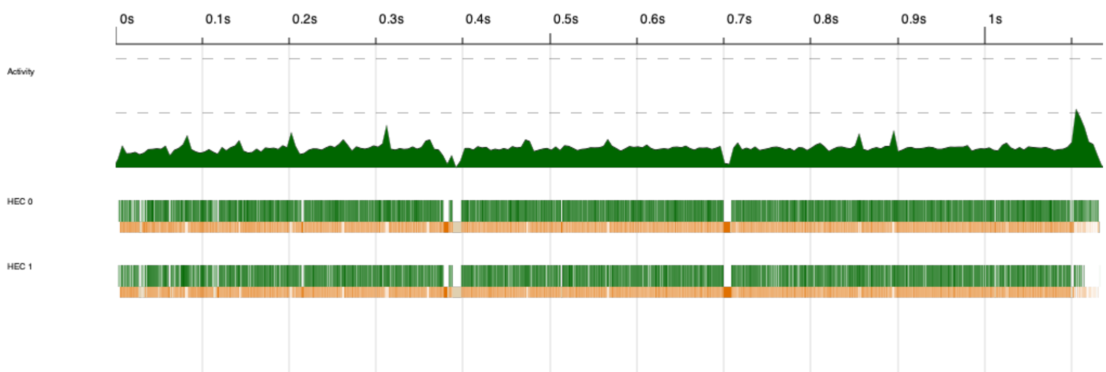

Fig 4. ipFreq -N2 threadscope

After the experiment and some search online, I realize that I can hardly see a speedup on a IO based MapReduce application, so I choose to solve another problem in this MapReduce framework.

3. 8-Puzzle analysis using parallel MapReduce

8-puzzle is a game on a 3*3 board and there are 9 numbers from 0 to 8 on it. The goal is to swap 0 and its neighbor to get to a final state. Below is an example on it:

```
   1  3       1     3       1  2  3       1  2  3       1  2  3
4  2  5   =>  4  2  5   =>  4     5   =>  4  5      =>  4  5  6
7  8  6       7  8  6       7  8  6       7  8  6       7  8

initial                                                  goal
```

I implement a search based on Manhattan heuristic to solve a single board. Each board can be presented as a list in Haskell. For example, the left most board on the picture is [0,1,3,4,2,5,7,8,6]. By using a set as a priority queue, I designed a BFS like algorithm to solve it in a quick fashion.

Although the problem itself is simple, it's not easy to compute a lot of them and get their result efficiently. A parallel MapReduce should be able to speed the process up and apply some analysis on the result in the reduce step.

One thing worth investigating is for all solvable boards, what's the statistic of steps used. BFS can find the shortest solution while a heuristic based search can't guarantee this. Getting the full stats on steps used provides a sense of how many more steps do a Manhattan heuristic uses on 8-puzzle problem.

To generate all possible initial states we need Data.List.permutations. 8-puzzle is solvable only when the list has even inversions, so we apply this filter condition and get 9!/2 = 181440 solvable boards. In the map stage, each mapper solves one boards, mapping each board to a pair (steps, 1). The shuffle stage we group the pairs by key and made the value a list of ones. The reduce stage simply get the sum or length of the list. The final output will be a key value pair list, where key is steps count and value is how many boards are solved using that many steps. This histogram like stats can show the distribution of steps using Manhattan heuristic. I also tried running a normal BFS for comparison. Unfortunately, the naive BFS is too slow to get any meaningful result.

Here's the performance test on running 1000 boards in the MapReduce framework and sequential version.

```
INIT    time    0.001s  (  0.004s elapsed)
MUT     time    9.593s  (  9.687s elapsed)
GC      time    1.530s  (  1.570s elapsed)
EXIT    time    0.000s  (  0.013s elapsed)
Total   time   11.123s  ( 11.274s elapsed)

Alloc rate    3,837,988,325 bytes per MUT second

Productivity  86.2% of total user, 85.9% of total elapsed
```

Fig 5. puzzleSovler -N1

```
INIT    time    0.001s  (  0.004s elapsed)
MUT     time    8.299s  (  5.808s elapsed)
GC      time    5.259s  (  1.094s elapsed)
EXIT    time    0.000s  (  0.011s elapsed)
Total   time   13.559s  (  6.918s elapsed)

Alloc rate    4,437,876,386 bytes per MUT second

Productivity  61.2% of total user, 84.0% of total elapsed
```

Fig 6. puzzleSovler -N2

```
INIT     time     0.001s  (  0.003s elapsed)
MUT      time     9.511s  (  9.604s elapsed)
GC       time     1.519s  (  1.557s elapsed)
EXIT     time     0.000s  (  0.004s elapsed)
Total    time    11.031s  ( 11.169s elapsed)

Alloc rate     3,870,870,834 bytes per MUT second

Productivity  86.2% of total user, 86.0% of total elapsed
```

Fig 7. puzzleSeq

Here are the threadscope diagrams for -N1 and -N2 option:

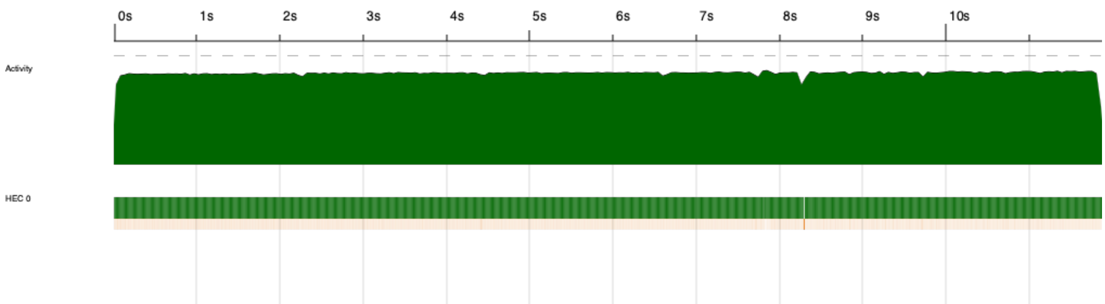

Fig 8. puzzleSovler -N1 threadscope

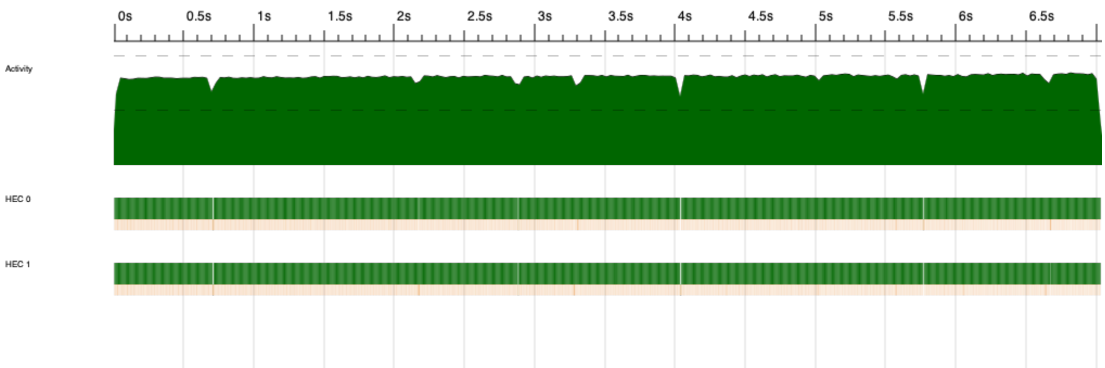

Fig 9. puzzleSovler -N2 threadscope

The performance test shows that this problem speeds up a lot by using parallel strategy. The speedup factor is about 11.2 / 6.9 = 1.6. Although ideally we should gain a factor near 2, 1.6 is still a decent speedup consider we do have sequential steps.

4. Conclusion

Parallel MapReduce doesn’t guarantee a speedup on some applications: If an application is IO heavy and has a sequential bottleneck, we can’t see a speedup in the performance test. A lot of real world MapReduce applications works fine because it has large enough input states and the tasks are assigned to machines. In a single node multiple cores scenario, this kind of application will suffer from IO and GC. In the book Real World Haskell, one way to solve this is using ByteString to optimize the IO and using a large text file to analyze (248 MB). To observe a speedup directly, we have to choose a computation heavy task like some search problem.

5. Code list

My project include these files:

mapreduce.hs: mapreduce module
ipFreq.hs: count ip visit frequency in parallel
ipSeq.hs: count ip visit in sequence
puzzleSolver.hs: parallel mapreduce on 8-puzzle solving
puzzleSeq.hs: sequential mapreduce on 8-puzzle solving
puzzle.hs: module for solving 8-puzzle
readme.txt: instruction on how to compile and run the code
access.log.txt: input file of ipFreq

References

1.Parallel and Concurrent Programming in Haskell, Simon Marlow
2.MapReduce as a Moand, Julian Porter
3.Real World Haskell,Bryan O'Sullivan, Don Stewart, and John Goerzen Chapter 24.

Appendix: Source Code
As sequential version is trivial, they are not listed.

puzzle.hs

```
module Puzzle
  ( solve,
  )
where

import Data.Array
import Data.Maybe
import qualified Data.Set as S

data Puzzle = Puzzle (Array (Int, Int) Int) deriving (Eq, Ord)

data State = State (S.Set (Int, Puzzle, [Int])) (S.Set Puzzle)

finalState = Puzzle $ listArray ((0, 0), (2, 2)) $ [1, 2, 3, 4, 5, 6, 7, 8, 0]

-- get a number's index on the board
getIndex :: Int -> Puzzle -> (Int, Int)
getIndex n (Puzzle p) = head $ filter (\idx -> p ! idx == n) $ indices p

-- get neighbors of zero
getNeighbors :: Puzzle -> [(Int, Int)]
getNeighbors (Puzzle p) = filter (`elem` indices p) [(zx -1, zy), (zx + 1, zy), (zx, zy -1), (zx, zy +
1)]
  where
    (zx, zy) = getIndex 0 (Puzzle p)

-- swap a pos with 0
swap :: Puzzle -> (Int, Int) -> (Int, Puzzle)
swap (Puzzle p) pos = (p ! pos, Puzzle $ p // [((zx, zy), p ! pos), (pos, 0)])
  where
    (zx, zy) = getIndex 0 (Puzzle p)

-- possible next moves
getMoves :: Puzzle -> [(Int, Puzzle)]
getMoves (Puzzle p) = map (swap (Puzzle p)) $ getNeighbors (Puzzle p)

-- manhattan heurisitc for current board
manhattanSum :: Puzzle -> Int
```

```haskell
manhattanSum p = sum $ map manhattan [0 .. 8]
  where
    manhattan num = abs (fx - x) + abs (fy - y)
      where
        (fx, fy) = getIndex num finalState
        (x, y) = getIndex num p

transfer :: State -> (Puzzle, [Int], State)
transfer (State queue visited) = (puz, moves, State nextqueue (S.insert puz visited))
  where
    ((h, puz, moves), curqueue) = fromJust $ S.minView queue
    nextmoves = S.fromList $ filter (\(_, p) -> p `S.notMember` visited) $ getMoves puz
    nextqueue = curqueue `S.union` (S.map (\(moved, p) -> (manhattanSum p, p, moved :
moves)) nextmoves)

search :: Int -> State -> Int
search i curstate
  | p == finalState = length moves
  | otherwise = search (i + 1) nextState
  where
    (p, moves, nextState) = transfer curstate

searchDebug :: Int -> State -> [Int]
searchDebug i curstate
  | p == finalState = moves
  | otherwise = searchDebug (i + 1) nextState
  where
    (p, moves, nextState) = transfer curstate

solve :: [Int] -> Int
solve l = search 0 start
  where
    start = State (S.singleton (manhattanSum p, p, [])) S.empty
      where
        p = Puzzle $ listArray ((0, 0), (2, 2)) l

solveDebug :: [Int] -> [Int]
solveDebug l = searchDebug 0 start
  where
    start = State (S.singleton (manhattanSum p, p, [])) S.empty
      where
        p = Puzzle $ listArray ((0, 0), (2, 2)) l
```

ipfreq.hs

```haskell
import System.Environment(getArgs)
import System.IO(readFile)
import System.Exit(exitFailure)
import Data.List(sortBy)
import MapReduce (mapReduce)
import Control.Parallel.Strategies
import Control.Parallel (pseq)
import Control.DeepSeq
import Control.Exception
import Data.List
import Data.Ord
import Data.Function (on)


mapper :: String -> (String, Int)
mapper w = (w, 1)

shuffler :: (Eq a) => [(a,b)] -> [(a,[b])]
shuffler = map (\x -> (fst $ head x,  map snd x)) . groupBy ((==) `on` fst)

reducer :: (String, [Int]) -> (String, Int)
reducer (w, l)  = (w, (sum l))

parse :: String -> String
parse w = head (words w)

main :: IO ()
main = do args <- getArgs
          case args of
            [filename] -> do
              text <- readFile filename
              let linelist = lines text
              let dict = map parse linelist
              let mr = mapReduce mapper shuffler reducer dict
              let result = sortBy (\(_ , cnt) (_ , cnt') -> compare cnt' cnt) mr
              print result
```

mapreduce.hs

```haskell
module MapReduce
   (
     mapReduce
   ) where

import Control.Parallel (pseq)
import Control.Parallel.Strategies

mapReduce :: (NFData a, NFData b, NFData c, NFData d) =>
   (a -> b) -- mapper
   -> ([b] -> [c]) -- shuffle
   -> (c -> d)    -- reducer
   -> [a] -- state
   -> [d] -- result

mapReduce mapFunc shuffleFunc reduceFunc input = mapResult `pseq` reduceResult
  where mapResult = parMap rdeepseq mapFunc input
        shuffleResult = shuffleFunc mapResult
        reduceResult = parMap rdeepseq reduceFunc shuffleResult
```

puzzlesolver.hs

```haskell
import System.Environment(getArgs)
import System.IO(readFile)
import System.Exit(exitFailure)
import Data.List(sortBy)
import MapReduce (mapReduce)
import Control.Parallel.Strategies
import Control.Parallel (pseq)
import Control.DeepSeq
import Control.Exception
import Data.List
import Data.Ord
import Data.Function (on)
import Puzzle (solve)
```

```haskell
shuffler :: Ord a => [(a,b)] -> [(a,[b])]
shuffler = map (\x -> (fst $ head x,  map snd x)) . groupBy ((==) `on` fst) . sortBy (comparing fst)

mapper :: [Int] -> (Int, Int)
mapper l = (solve l, 1)

reducer :: (Int, [Int]) -> (Int, Int)
reducer (t, l)  = (t, (sum l))

inversions :: [Int] -> Int
inversions [] = 0
inversions (x:xs) = (length (filter (<x) xs)) + inversions xs
main = do
	let per = permutations [1,2,3,4,5,6,7,8,0]
	let solvable = take 1000 (filter (\l -> inversions l `mod` 2 == 0) per)
	let res = mapReduce mapper shuffler reducer solvable
	print res
```