Parallel N-Puzzle Solver in Haskell

Zhonglin Yang (zy2496), Yuxuan Luo(yl4524)

2021/12/22

1. Introduction

N-Puzzle is the general name of category of puzzle games with $\sqrt{N+1}$ by $\sqrt{N+1}$ frames and $N$ sliding tiles inside the frame. For instance, an 8-Puzzle is played with a 3×3 frame and 8 tiles, numbered 1 through 8, placed inside the frame, leaving one empty slot unoccupied. The player is then able to slide adjacent tiles into the unoccupied slot until the goal state is reached (e.g., the board is solved). The goal of this game is to turn the scrambled start state into the goal state in as few moves as possible. The illustration below shows one solvable combination of the start state and the goal state of an 8-Puzzle game.

![Start State](image1.png) ![Goal State](image2.png)

2. Solving N-Puzzle

Finding the optimal solutions for N-Puzzles is an NP-Hard search problem. For any generic BFS algorithm, the number of states in its queue quadruples after each iteration, giving it an exponential run time complexity as it searches through $(N + 1)!$ different possible states of the frame. According to [1], an optimal 15-Puzzle solver using BFS can reach upwards to a depth of 80 with billions of intermediate frame states. For this reason, our implementable of the N-Puzzle Solver seeks a solution that is fast to compute as opposed to one that has the fewest moves.

3. Implementation

3.1 Algorithm

Our “best first search” N-Puzzle solver uses A* search in combination with Manhattan Distance to conduct informed searches within all possible moves.
Our A* search takes both the state of a next frame and the state of the search space into consideration. In our case, the resulting frame of any valid move is given a score of \( g(x) + h(x) \). \( g(x) \) is the cost, measured using Manhattan Distance, from the start state to the potential next state, while \( h(x) \) is the cost, also in Manhattan Distance, from the next state to the goal state. We then add each next state along with its cost to a min-priority queue where A* search chooses the state with the lowest cost from.

### 3.2 Data Structures and Representations

We are using a 2d-array of Int to represent the frame (referred to as grid in the code) 

```haskell
type Grid = [[Int]]
```

We are using a min priority queue for (cost, candidate). Each candidate is an array of Grids from the start to the next state of the grid.

```haskell
import qualified Data.PQueue.Prio.Min as PQ
PQ.MinPQueue Int [Grid]
```

We are using a set of grids to mark the states we’ve already visited to avoid loops.

```haskell
import qualified Data.HashSet as S
S.Set Grid
```

### 3.3 Sequential Implementation

```haskell
min-pq <- [(start_state, 0)]
while (min-pq is not empty):
    cur_state <- min-heap.pop()
    if (cur_state.state != goal_state):
        for new_state in getNeighbors(cur_state):
            if new_state not in visited:
                min-pq.push( (new_state, getCost(new_state)) )
    else:
        return path_to(cur_state)
raise exception
```

Some simplifications are made here. In particular, any state mentioned are in fact [Grid] where the first element in [Grid] is the next state we are going to visit, and the last element in [Grid] is the start state.

### 3.4 Parallel Implementation Attempt #1

```haskell
getNextNodes goal d st l xss = PQ.fromList $ zip costs neighbors where
    costs = parMap rseq (getCost st d l goal) neighbors
    neighbors = parMap rpar (:xss) (getNeighbors $ head xss)
```
For our initial attempt, we decided to spark the evaluation of the neighbors and costs in parallel using \texttt{rpar} and \texttt{rseq}. In addition, we also planned to start evaluation of all neighbors instead of adding them to a priority queue.

### 3.5 Parallel Implementation Attempt #2

For our second attempt, we decided to solve multiple N-Puzzles in parallel using different evaluation strategies including static partitioning and dynamic partitioning. To get a better evaluation on the performance parallel computation, we eliminated unnecessary sequential computation of our program. Only N-puzzle solutions’ time complexities are printed at the end.

We started attempt #2 by changing \texttt{main.hs} file to make the solver solve multiple N-puzzles at the same time.

```haskell
main :: IO ()
main = do
  args <- checkArgs =<< getArgs
  content <- readFile $ args !! 0
  let input_list = splitOn "#" content
      grids = map inputToGrid input_list
      solution = map solveNPuzzle grids
  print (length solution) >> mapM_ print solution
```

For the sequential evaluation approach, we simply called \texttt{map solveNPuzzle} on \texttt{grids}. Then, we implemented two parallel evaluation approaches. The first approach is static partitioning. We split \texttt{grids} to two lists \texttt{as'} and \texttt{bs' } and used \texttt{rpar} and \texttt{force} to solve N-puzzles.

```haskell
solution = runEval $ do
  as' <- rpar (force (map solveNPuzzle as))
  bs' <- rpar (force (map solveNPuzzle bs))
  rseq as'
  rseq bs'
  return (as'++ bs')
```

When using static partitioning, we parallelly solve two lists of N-puzzles. We can only take advantage of two cores of CPU. To further evaluate parallel solving N-puzzles with more cores, we also implemented dynamic partitioning with \texttt{parList}.

```haskell
parMap :: (a -> b) -> [a] -> Eval [b]
parMap _ [] = return []
parMap f (a:as) = do b <- rpar (f a)
                     bs <- parMap f as
                     return b:bs
```
\text{return } (b:bs) \\
solution = \text{runEval (parMap solveNPuzzle grids)}

We used two set of N-puzzles to evaluate the two parallel approaches. The first set of N-puzzles is 1000 8-puzzles, and the second set is 40 15-puzzles. 8-puzzles are easy to solve, with most of the solutions’ time complexities are within 1000. 15-puzzles are much harder, with some of the 15-puzzles’ solutions exceeding 200000. We would like to find if the complexity of each puzzle can affect the program.

4. Performance Testing & Results

4.1 Results from Attempt #1

The initial attempt to construct \textit{next state} and their cost in parallel only produced negative performance improvement across our tests. We believe that our strategy did not work out due to the limitation of setup of N-Puzzle. Since each puzzle only has a maximum of 4 \textit{next states}, constructing them in parallel adds too much overhead in comparison to the computation done. Additionally, getCost uses Manhattan Distance which requires very little calculation. We suspect if the cost function is more complicated, we might start seeing performance gains by evaluating them in parallel.

Without parallelization, done in 3.23s.

With parallelization and -N4, done in 3.77s.
Performance degradation as more threads are used

Our attempt to evaluate every next states as soon as they are defined also failed. Due to the nature of the search algorithm, we needed to maintain a global state of visited states visible to all parallel evaluations so they don’t repeat the same work. Because there is no easy way to accomplish that in Haskell, we decided to proceed without it. In practice, we found this modification brings no improvement over sequential evaluation for easy puzzles, again due to the overhead. We also found that this modification breaks the program when it is running on slightly more difficult puzzles, due to the amount of repeated work in parallel stalling the system.

4.2 Results from Attempt #2

The evaluation results of solving 1000 8-puzzles with different parallel approaches are as following.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential</td>
<td>6.438s</td>
<td>1x</td>
</tr>
<tr>
<td>static partitioning -N2</td>
<td>3.419s</td>
<td>1.88x</td>
</tr>
<tr>
<td>dynamic partitioning -N2</td>
<td>4.429s</td>
<td>1.45x</td>
</tr>
<tr>
<td>dynamic partitioning -N4</td>
<td>3.044s</td>
<td>2.11x</td>
</tr>
<tr>
<td>dynamic partitioning -N8</td>
<td>2.399s</td>
<td>2.68x</td>
</tr>
</tbody>
</table>

We could see the static partitioning approach has a pretty good performance with 2 cores. The speedup is 1.88x which is close to 2. By contrast, the dynamic partitioning approach with 2 cores doesn’t perform comparably good. Its speedup is only 1.45x. When 4 cores and 8 cores are used, the speedup increase to 2.11x and 2.68x. It seems that if we want the best performance of solving multiple N-puzzles, we can just use more cores with dynamic partitioning. However, when evaluating the three approaches on solving 15-puzzles, we found it is not the case.

The evaluation results of solving 40 15-puzzles with different parallel approaches are as following.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential</td>
<td>177.366s</td>
<td>1x</td>
</tr>
</tbody>
</table>
From the results, we could intuitively see that 15-puzzle is much harder to solve. The average time to solve one 15-puzzle is around 700x of the time to solve. When N-puzzles are much harder to solve, dynamic partitioning still can help accelerate the solver program. With 2 cores and 4 cores, the speedups are better than the speedups on solving 8-puzzles. However, when 8 cores are used, the performance declines a lot. It takes extra 15 seconds to finish solving 15-puzzles when using extra 4 cores. Thus, we took a look at the eventlog on threadscope.

On the eventlog, we could see the unusual patterns of several time intervals where the activity becomes zero and CPU cores are busy to deal with garbage collection. And along the program running, each time the pattern appears, the time interval become larger. At around 65 seconds, several cores start idling. Several intervals of garbage collection and several CPU cores idling after 65 seconds cause the program which utilize 8 cores runs slower than the program which utilize 4 cores.

We inspected some other metrics on eventlog and found when using 8 cores to solving 40 15-puzzles, the maximum heap size increases to 5.9 GiB which is 3 times
of the maximum heap size when using 4 cores. Large heap probably causes the slowdown.

The unusual slowdown of using 8 cores to solve 15-puzzles suggests us that to solve computation heavy problems, it is not always true that utilizing more resources can lead to a better result. We need to know that some unexpected things can emerge and can become counterproductive.

Code:

```haskell
module Main where
import GHC.Conc(par)
import Control.Parallel.Strategies hiding(parMap)
import Control.DeepSeq
import Data.List.Split
import System.Environment (getArgs)
import System.Exit (exitSuccess)
import Grid (getSolvedGrid, Grid)
import Solver (solve, solve')
import Logger
import Parser

checkArgs :: [String] -> IO [String]
checkArgs xs = if null xs then displayHelp >> exitSuccess else pure xs

inputToGrid :: String -> [[Int]]
inputToGrid input = transformToGrid (words <$> (drop 1 . clearInput . lines $ input))

solveNPuzzle :: Grid -> Int
solveNPuzzle grid = solve' (getSolvedGrid $ length grid) grid (Nothing, Nothing)

parMap :: (a -> b) -> [a] -> Eval [b]
parMap _ [] = return []
parMap f (a:as) = do b <- rpar (f a)
                      bs <- parMap f as
                      return (b:bs)

main :: IO ()
main = do
         args <- checkArgs =<< getArgs
         content <- readFile $ args !! 0
         let input_list = splitOn "#" content
             grids = map inputToGrid input_list
                     (map solveNPuzzle grids)
```
{- runEval (parMap solveNPuzzle grids) -}
(as, bs) = splitAt (length grids `div` 2) grids
solution = runEval $ do
    as' <- rpar (force (map solveNPuzzle as))
    bs' <- rpar (force (map solveNPuzzle bs))
    rseq as'
    rseq bs'
    return (as' ++ bs')
print (length solution) >> mapM_ print solution

module Solver (SearchType(..), readSearchType, solve') where
import qualified Data.PQueue.Prio.Min as PQ
import qualified Data.HashSet as S
import Logger
import Grid
import Distance

data SearchType = Astar | Uniform | Greedy deriving Eq
type NextNodesFunc = Int -> [Grid] -> PQ.MinPQueue Int [Grid]

instance Show SearchType where
    show Astar      = "A*
    show Uniform    = "Uniform cost"
    show Greedy     = "Greedy"

readSearchType :: String -> Maybe SearchType
readSearchType s = case s of
    "astar" -> Just Astar
    "uniform" -> Just Uniform
    "greedy" -> Just Greedy
    _ -> Nothing

-- Returns the cost of a node according to the SearchType currently used
getCost :: SearchType -> Distance -> Int -> Grid -> Grid -> Int
getCost st d l grid goal = let dist = calcDistance d grid goal in case st of
    Astar      -> dist + 1  -- A*: h cost + g cost
    Uniform    -> l           -- Uniform: g cost only
    Greedy     -> dist        -- Greedy: h cost only

-- Returns a PQueue containing the next nodes (value + cost)
getNextNodes :: Grid -> Distance -> SearchType -> Int -> [Grid] -> PQ.MinPQueue Int [Grid]
getNextNodes goal d st l xss = PQ.fromList $ zip costs neighbors where
    costs = (getCost st d l goal) <$> head <$> neighbors
neighbors = map (:xss) $ getNeighbors $ head xss

-- goal : stage to reach ; xss : path from begining to current node ; os : open set ; cs : close set ; nn : nextNodes function ; n : time complexity ; m : space complexity ; l : xss length
runSearch' :: Grid -> [Grid] -> PQ.MinPQueue Int [Grid] -> S.Set Grid ->
NextNodesFunc -> Int -> Int -> Int -> Int
runSearch' goal xss os cs nn n m l
    | head xss == goal = n
    | suc /= PQ.empty = runSearch' goal ((minim suc):xss) (PQ.union os $ PQ.deleteMin suc) cs' nn (n+1) size (l+1)
    | suc == PQ.empty && os /= PQ.empty = runSearch' goal (tail xss) os cs' nn (n+1) size (l-1)
    | otherwise = 0 where
    suc = PQ.filter (\x -> S.notMember (head x) cs) $ nn l xss
    cs' = S.insert (head xss) cs
    minim x = head . snd $ PQ.findMin x
    size = if PQ.size os > m then PQ.size os else m

solve' :: Grid -> Grid -> (Maybe SearchType, Maybe Distance) -> Int
solve' goal xs (Nothing, Nothing) = runSearch' goal [xs] PQ.empty S.empty
( getNextNodes goal defaultHeuristic defaultSearch ) 0 0 1
solve' goal xs (Just st, Nothing) = runSearch' goal [xs] PQ.empty S.empty
( getNextNodes goal defaultHeuristic st ) 0 0 1
solve' goal xs (Nothing, Just d) = runSearch' goal [xs] PQ.empty S.empty
( getNextNodes goal d defaultSearch ) 0 0 1
solve' goal xs (Just st, Just d) = runSearch' goal [xs] PQ.empty S.empty
( getNextNodes goal d st ) 0 0 1

import Distance
import Solver

maxPuzzle :: Int
maxPuzzle = 999

-- Strips comments and empty lines
clearInput :: [String] -> [String]
clearInput xs = filter (/="") $ map (head . splitOn "#") xs

-- Checking size of the puzzle
isValidSize :: [[Int]] -> Bool
isValidSize xss = let ys = map (`^2`) [3..maxPuzzle] in (length . concat) xss `elem` ys
-- Sorts input and compare it to an enum list of the same size
hasValidContent :: [[Int]] -> Bool
hasValidContent xss = let n = length xs - 1; xs = concat xss; ys = [0..n]
in sort xs == ys

-- Returns the input as [[Int]] if it is valid, otherwise returns Nothing
transformInput :: [[String]] -> Maybe [[Int]]
transformInput xss
  | all isDigit (concat $ concat xss) == False = Nothing
  | otherwise = let xss' = (map read) <$> xss in if isValidSize xss' &&
    hasValidContent xss' then Just xss' else Nothing

transformToGrid :: [[String]] -> [[Int]]
transformToGrid xss =
  let xss' = (map read) <$> xss in xss'

-- Check programs args, and returns associated flags
parseArgs :: [String] -> (Maybe SearchType, Maybe Distance)
parseArgs xs = case length xs of
  1   -> (Nothing, Nothing)
  2   -> (readSearchType (xs !! 1), Nothing)
  _   -> (readSearchType (xs !! 1), readDistance (xs !! 2))