PFP Final Project Report
Wendy Wang (www2105), Raymond Li (rwl2117)

Project Description
- What we implemented and how
  - All-pairs shortest distance problem
    - Find the shortest distance between all possible pairs of nodes in a graph using Dijkstra’s algorithm for shortest path
    - Parallelization component: calculate distance between pairs in the graph in parallel rather than sequentially to reduce runtime
  - The executable takes in an integer \( n \), randomly generates a graph with \( n \) nodes, then outputs a calculation of all-pairs for the graph

Performance Figures:

Single-Threaded

$ time ./final 200 +RTS -N1 -ls

real 0m17.106s
user 0m24.718s
sys 0m2.136s

Parallelized w/ Four Cores

$ time ./final 200 +RTS -N4 -ls

real 0m5.705s
user 0m20.116s
sys 0m0.466s

Speedup = user/real = 20.116/5.705 = 3.53

Full Code Listing

import           Data.Graph
import           Data.IntMap.Lazy
import           Data.Set                  ( randomRIO
import           System.Random                  , getStdRandom
import           Control.Monad                  , randomR
import           Data.List                      ( replicateM )
import           Control.Parallel.Strategies    ( nub )
import           System.Environment             ( parMap
import           System.Exit                    , rdeepseq)
import

{-
   MUST INSTALL PACKAGES:
   -- $ cabal install --lib random
   $ stack install parallel

References:

main :: IO ()
main = do
  args <- getArgs
  case args of
    [strNum] -> do
      let numNodes = read strNum :: Int
          (g, intMapArr) <- allPairsParallel numNodes
      putStrLn "Graph:
      print g
      putStrLn "IntMap:
      mapM_ print intMapArr
    _ -> do
        pn <- getProgName
        die $ "Usage: " ++ pn ++ " <number of nodes in random graph>"

generateTestGraph :: Graph
generateTestGraph = buildG (0, 6) [(0, 1), (2, 3), (5, 1), (6, 1),
(5, 4), (1, 3), (4, 5), (3, 6)]

generateTestGraphTwo :: Graph
generateTestGraphTwo = buildG (0, 9) [(0, 2), (1, 0), (1, 3), (2, 0),
(3, 8), (3, 9), (4, 3), (4, 8), (5, 6), (6, 3), (6, 4), (7, 1), (7,
2), (7, 5), (8, 3), (9, 1)]

allPairsParallel :: Int -> IO (Graph, [IntMap Double])
allPairsParallel numNodes = do
  g <- randGraph numNodes
  let distanceMaps = computeDistancesParallel [0 .. numNodes] g
  return (g, distanceMaps)
computeDistancesParallel :: [Int] -> Graph -> [IntMap Double]
computeDistancesParallel [] _ = []
computeDistancesParallel l g = parMap rdeepseq (dijkstra g) l

allPairs :: Int -> IO (Graph, [IntMap Double])
allPairs numNodes = do
g <- randGraph numNodes
let distanceMaps = computeDistances [0 .. numNodes] g
return (g, distanceMaps)

computeDistances :: [Int] -> Graph -> [IntMap Double]
computeDistances [] _ = []
computeDistances (n : ns) g = dijkstra g n : computeDistances ns g

callDijkstra :: Int -> Int -> IO (Graph, IntMap Double)
callDijkstra numNodes startNode = do
g <- randGraph numNodes
let distanceMap = dijkstra g startNode
return (g, distanceMap)

callGraph :: Int -> IO Graph
callGraph n = do
edgs <- allEdges n
return $ buildG (0, n) edgs

allEdges :: Int -> IO [Edge]
allEdges n = do
listOfEdges <- sequence $ zipWith randEdges [0 .. n] (replicate (n + 1) n)
let allEdgeList = concat listOfEdges
return allEdgeList

randEdges :: Int -> Int -> IO [Edge]
randEdges currNode numNodes = do
randInts <- randIntList numNodes
return [ (currNode, m) | m <- randInts, currNode /= m ]

randIntList :: Int -> IO [Int]
randIntList n = do
deps <- numDependecies n
ints <- replicateM deps $ randomRIO (0, n)
return $ nub ints
numDependencies :: Int -> IO Int
numDependencies n = do
  r <- getStdRandom (randomR (1, n))
  return r

dijkstra :: Graph -> Int -> IntMap Double
dijkstra g start = dijkstraHelper g distanceMap visitedSet
  where
distanceMap = initializeIntMap start (vertices g)
visitedSet = initializeSet
dijkstraHelper :: Graph -> IntMap Double -> Set Int -> IntMap Double
dijkstraHelper g distances visitedSet = case smallestDistanceNode of
  Nothing -> distances
  Just _ -> dijkstraHelper g
    updatedDistanceMap
    (addToSet visitedSet unwrappedSmallestDistanceNode)
  where
    smallestDistanceNode =
      getSmallestDistanceNode (vertices g) visitedSet distances
    Just unwrappedSmallestDistanceNode =
      getSmallestDistanceNode (vertices g) visitedSet distances
    updatedDistanceMap = updateDistanceMap
      unwrappedSmallestDistanceNode
      (getAdjacentVertices g unwrappedSmallestDistanceNode) distances

getSmallestDistanceNode :: [Int] -> Set Int -> IntMap Double -> Maybe Int
getSmallestDistanceNode q visited distances = case validNodes of
  [] -> Nothing
  _ -> Just
    $ Prelude.foldl (\x y -> getCloserNode x y distances) firstElem
    validNodes
  where
    validNodes = Prelude.filter (\x -> Data.Set.notMember x visited) q
    firstElem = head validNodes

gETCHARGER:: Int -> Int -> IntMap Double -> Int
getCloserNode a b m = case (compare l r) of
  LT -> a
  EQ -> a
GT -> b
where
  Just l = Data.IntMap.Lazy.lookup a m
  Just r = Data.IntMap.Lazy.lookup b m

getAdjacentVertices :: Graph -> Int -> [Int]
getAdjacentVertices g v = convertEdgeToOutNodes $ getEdgesOutOf g v

getEdgesOutOf :: Graph -> Int -> [Edge]
getEdgesOutOf g i = Prelude.filter (\(s, _) -> s == i) $ edges g

convertEdgeToOutNodes :: [Edge] -> [Int]
convertEdgeToOutNodes [] = []
convertEdgeToOutNodes ((_, o) : xs) = o : convertEdgeToOutNodes xs

initializeIntMap :: Int -> [Int] -> IntMap Double
initializeIntMap s l = adjust f s
  (Data.IntMap.Lazy.fromList [ (x, 1 / 0) | x <- l ])
  where f _ = 0

initializeSet :: Set Int
initializeSet = Data.Set.fromList []

addToSet :: Set Int -> Int -> Set Int
addToSet s i = Data.Set.insert i s

updateDistanceMap :: Int -> [Int] -> IntMap Double -> IntMap Double
updateDistanceMap _       []       m = m
updateDistanceMap closest (a : as) m = updateDistanceMap closest
  as
where
  Just closestDistance = Data.IntMap.Lazy.lookup closest m
  updatedDistanceMap   = updateDistanceNode closestDistance a m

updatedDistanceMap
where
  Just closestDistance = Data.IntMap.Lazy.lookup closest m
  updatedDistanceMap   = updateDistanceNode closestDistance a m

updateDistanceNode :: Double -> Int -> IntMap Double -> IntMap Double
updateDistanceNode closestDistance adj m =
  case (compare (closestDistance + 1) adjDistance) of
    LT -> updatedMap
    EQ -> updatedMap
GT -> m
where
  Just adjDistance = Data.IntMap.Lazy.lookup adj m
  f _ = closestDistance + 1.0
  updatedMap = adjust f adj m