Blocked Floyd-Warshall Algorithm

Pelin Cetin (pc2807)

I. INTRODUCTION

In a directed weighted graph with positive or negative edge weights, the Floyd–Warshall method is used to identify the shortest pathways, or rather summed weights, between all pairs of vertices in a single iteration of the algorithm.

With a runtime of V^3 where V is a vertex, the normal Floyd–Warshall algorithm examines all potential pathways across the graph between each pair of vertices. Every edge combination is put to the test. It accomplishes its goal of finding the shortest path for all pairs by gradually refining a prediction of the shortest path between two vertices until the prediction is ideal.

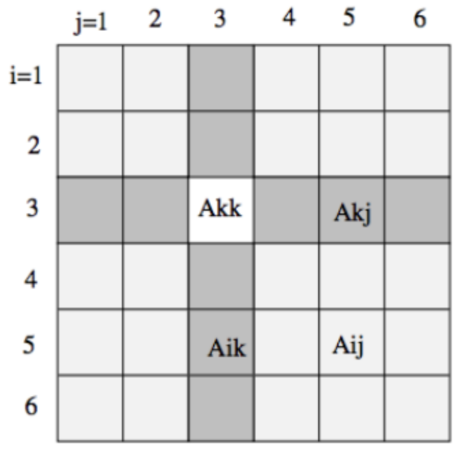
The blocked Floyd-Warshall algorithm has been created as an alternative to the normal Floyd-Warshall algorithm. This version is meant to be run in parallel and hence reduce the runtime. The algorithm splits the adjacency matrix into blocks and processes them in three different phases: dependent, partially dependent, and independent.

II. BLOCKED FLOYD WARSHALL ALGORITHM

The Floyd-Warshall algorithm is a dynamic programming algorithm, which means that the problem is broken down into smaller problems and is solved in a recursive manner. The blocked version of the algorithm is also dynamic.

The blocked Floyd-Warshall algorithm splits the adjacency matrix that is given to the program as input into blocks and processes them in three different phases: dependent, partially dependent, and independent.

First, in the dependent phase, the kth diagonal block is processed. This means that when k=0, the block on the top left corner is processed. When k=1, the block that is on the first row and first column is processed. The program keeps processing the kth block until all blocks on the diagonal of the block adjacency matrix are processed. The dependent phase cannot be parallelized, as the other phases depend on it. However, before we increment k, we go into the partially dependent phase. In this phase, we process the kth row and the kth column of blocks. This means that when k=0, this phase processes the zeroth column and zeroth row. Lastly, in the independent phase, the remaining blocks are processed.



[1] https://moorejs.github.io/APSP-in-parallel/

The term processed means that the floyd_warshall_in_place algorithm is called, which takes three matrices of size bxb, where b refers to the block size. This function ultimately does the comparison of [i][k] + [k][j] with [i][j] that is fundamental to the Floyd-Warshall algorithm.

III. IMPLEMENTATION

A. Adjacency Matrix Representation

The graph is represented as a list of data constructor Weight, which has been defined as either having an Integer value associated with it or having the value None. The value None refers to the state of one vertex not having a direct connection to another. Additionally, a diagonal line of Weight 0s runs throughout the input and output adjacency matrices since for example, the shortest path from vertex A to vertex A should be 0.

B. Random Adjacency Matrix Generation

The adjacency matrix, as explained above, should have a diagonal line of Weight 0s run throughout it. The remaining places should either have a weighted edge or be None. To come up with a random adjacency matrix, I have used the function randomIO from the module System.Random to generate a boolean and if it is false, I prepend None to the matrix. If it is true, I need to append a Weight that has a random integer associated with it. To make it random, I use another function from the System.Random module: randomRIO. I gave it a range of -10 to 200 and it will pick a random number. I then convert this Monad to an Int and create a Weight data type and prepend that to the matrix.

C. Sequential Implementation with the C Matrix

The reference material uses three matrices of bxb size for the floyd_warshall_in_place, as explained above. In this sequential version of the program, I essentially converted the C++ code into Haskell. However, it is important to note that this could not be parallelized in Haskell. The C++ code essentially accessed the memory places of various indexes of the adjacency matrix and updated them. This is tricky to do in Haskell due to the fact that objects are immutable and indexing does not exist. I had to create a function where if the list and the index is given, it returns the element. Moreover, the C++ implementation saved the new [i][j] values in the C matrix given as a parameter to the floyd_warshall_in_place function. In my implementation, I returned the C matrix and appended that to the first half of the input variable to create the output. Since all threads would return a C matrix and then it would be difficult to stitch them back together, I had to change the implementation. However, I still kept this file as a nod to the reference material.

D. Sequential Implementation without the C Matrix

Instead of returning the C matrix in floyd_warshall_in_place, I return the indexes where a more ideal path has been found as well as the new weight that should be there. I essentially do this by returning a list of tuples with the index and the new weight. After the functions have returned, I loop over this list and update the adjacency matrix.

E. Parallel Implementation

In order to make my program work in parallel, I called the partially dependent and inner independent phases with differing values of the for loop. I achieved this by first creating a list with the possible i values of the for loop. I then mapped different elements of this list as an argument to the phase functions and called spawnP on them. For example, if i starts from 0 and partially_dependent phase function takes i as an argument, I map 0 as an argument to the function and call it with spawnP. I also had to tweak the phase functions so that they would not be recursive themselves. Additionally, spawnP requires a pure expression. The partially_dependent and inner_independent phase functions return a list of Weights. Therefore, in order for my program to compile, I had to tweak the data constructor Weight to derive normal-form data as well. In order for Weight to work both as a normal-form data and not, I also had to derive Generics, a GHC function that essentially allows the developer to use the same code with differing data types.

IV. RESULTS

I am on a Macbook 2016, 3.1 GHz Dual-Core Intel Core i5. I ran Sequential_fw_block.hs and Parallel_fw_block.hs to get the results.

	Sequential	Parallel
64 vertices, block size = 8	316.21 secs	314.62 secs
70 vertices, block size = 7	462.78 secs	457.64 secs
80 vertices, block size = 8	958.63 secs	904.47 secs

V. REFERENCE

https://moorejs.github.io/APSP-in-parallel/

VI. SOURCE CODE

Graph.hs

```
{-# LANGUAGE DeriveAnyClass, DeriveGeneric #-}

module Graph where

import Control.Parallel
```

```haskell
import Control.Parallel.Strategies
import GHC.Generics (Generic)

data Weight = Weight Int | None deriving (Eq, Ord, Show, Generic, NFData)

addWeights :: Weight -> Weight -> Weight
addWeights (Weight x) (Weight y) = Weight (x + y)
addWeights _ _ = None

dataAt :: Int -> [Weight] -> Weight
dataAt _ [] = error "Empty List!"
dataAt y (x:xs)  | y <= 0 = x
                 | otherwise = dataAt (y-1) xs

removeItem :: Int -> [Int] -> [Int]
removeItem _ []                 = []
removeItem x (y:ys) | x == y    = removeItem x ys
                    | otherwise = y : removeItem x ys

replace_nth :: [Weight] -> (Int, Weight) -> [Weight]
replace_nth [] _ = []
replace_nth (_:xs) (0,a) = a:xs
replace_nth (x:xs) (n,a) = if n < 0 then (x:xs) else x: replace_nth xs (n-1,a)

replace_n_list :: [(Int, Weight)] -> [Weight] -> [Weight]
replace_n_list _ [] = []
replace_n_list [] input = input
replace_n_list (x:xs) input = replace_n_list xs (replace_nth input x)
```

Sequential_fw_block_with_C.hs

```haskell
{-
   Sequential Floyd-Warshall algorithm with 2-D block mapping in Haskell
   with the C matrix in floyd_warshall_in_place
-}

module Sequential_fw_block_with_C where

import Data.List
--import Debug.Trace
```

```haskell
data Weight = Weight Int | None deriving (Eq, Ord, Show)

addWeights :: Weight -> Weight -> Weight
addWeights (Weight x) (Weight y) = Weight (x + y)
addWeights _ _ = None

dataAt :: Int -> [Weight] -> Weight
dataAt _ [] = error "Empty List!"
dataAt y (x:xs)  | y <= 0 = x
                | otherwise = dataAt (y-1) xs

replace_nth :: [Weight] -> (Int, Weight) -> [Weight]
replace_nth [] _ = []
replace_nth (_:xs) (0,a) = a:xs
replace_nth (x:xs) (n,a) = if n < 0 then (x:xs) else x: replace_nth xs (n-1,a)

loops :: Int -> Int -> Int -> Int -> Int -> Int -> [Weight] -> [Weight] -> [Weight] ->
[Weight]
loops k n kth i j b l_a l_b l_c =
   if i == b then l_c
   else
       if j == b then loops k n kth (i + 1) 0 b l_a l_b l_c
       else
           if element > sum1 then loops k n kth i (j+1) b l_a l_b new_C
           else
               --trace (show element)
               --trace (show sum1)
               --trace (show l_c)
               loops k n kth i (j+1) b l_a l_b l_c
                   where element = dataAt (i*n + j) l_c
                         sum1 = addWeights (dataAt (i*n + k) l_a) (dataAt (kth + j)
l_b)
                         new_C = replace_nth l_c ((i*n + j), sum1)


floyd_warshall_in_place :: [Weight] -> [Weight] -> [Weight] -> Int -> Int -> Int ->
[Weight]
floyd_warshall_in_place l_a l_b l_c b n k =
   if k == b then l_c
   else
       floyd_warshall_in_place l_a l_b (loops k n kth 0 0 b l_a l_b l_c) b n (k+1)
           where kth = k*n
```

```haskell
inner_independent_phase :: Int -> Int -> Int -> Int -> Int -> [Weight] -> [Weight]
inner_independent_phase i j k b n input =
   if j == (n `div` b) then input
   else
       if j == k then inner_independent_phase i (j+1) k b n input
       else
           inner_independent_phase i (j+1) k b n res
           where
               l_a = drop (i*b*n + k*b) input
               l_b = drop (k*b*n + j*b) input
               (first_half, l_c) = splitAt (i*b*n + j*b) input
               res = first_half++(floyd_warshall_in_place l_a l_b l_c b n 0)


independent_phase :: Int -> Int -> Int -> Int -> [Weight] -> [Weight]
independent_phase i k b n input =
   if i == (n `div` b) then input
   else
       if i == k then independent_phase (i+1) k b n input
       else
           --trace (show input)
           independent_phase (i+1) k b n output
               where
                   l_a = drop (i*b*n + k*b) input
                   l_b = drop (k*b*n + k*b) input
                   (first_half, l_c) = splitAt (i*b*n + k*b) input
                   input_for_inner = first_half++(floyd_warshall_in_place l_a l_b l_c
b n 0)
                   output = inner_independent_phase i 0 k b n input_for_inner


partially_dependent_phase :: [Weight] -> Int -> Int -> Int -> Int -> [Weight]
partially_dependent_phase input j k n b =
   if j == (n `div` b) then input
   else
       if j == k then partially_dependent_phase input (j+1) k n b
       else
           --trace (show j)
           partially_dependent_phase (first_half++res) (j+1) k n b
               where
                   l_a = drop (k*b*n + k*b) input
                   l_b = drop (k*b*n + j*b) input
                   (first_half, l_c) = splitAt (k*b*n + j*b) input
                   res = floyd_warshall_in_place l_a l_b l_c b n 0

dependent_phase :: Int -> Int -> Int -> [Weight] -> [Weight]
dependent_phase k b n input =
   if k == (n `div` b) then input
   else
       dependent_phase (k + 1) b n new_in_output
           where
               l_a = drop (k*b*n + k*b) input
               l_b = drop (k*b*n + k*b) input
               (first_half, l_c) = splitAt (k*b*n + k*b) input
               new_dep_output = floyd_warshall_in_place l_a l_b l_c b n 0
               new_part_output = partially_dependent_phase
(first_half++new_dep_output) 0 k n b
               new_in_output = independent_phase 0 k b n new_part_output

floyd_warshall_blocked :: [Weight] -> Int -> Int -> [Weight]
floyd_warshall_blocked input n b =
   dependent_phase 0 b n input
```

Sequential_fw_block.hs

```haskell
{- Sequential Floyd-Warshall algorithm with 2-D block mapping in Haskell -}

module Sequential_fw_block (
       floyd_warshall_blocked
   ) where

import Control.Monad
import Data.List
import Graph
import System.Random(randomIO, randomRIO)

loops :: Int -> Int -> Int -> Int -> Int -> Int -> [Weight] -> [Weight] -> [Weight] ->
[(Int, Weight)] -> Int -> [(Int, Weight)]
loops k n kth i j b l_a l_b input replaced c_index =
   if i == b then replaced
   else
       if j == b then loops k n kth (i + 1) 0 b l_a l_b input replaced c_index
       else
```

```
           if element > sum1 then loops k n kth i (j+1) b l_a l_b input new_replaced
c_index
           else
               loops k n kth i (j+1) b l_a l_b input replaced c_index
                   where element = dataAt (c_index + (i*n + j)) input
                         sum1 = addWeights (dataAt (i*n + k) l_a) (dataAt (kth + j)
l_b)
                         new_replaced = ((c_index + (i*n + j)), sum1):replaced


floyd_warshall_in_place :: [Weight] -> [Weight] -> [Weight] -> Int -> Int -> Int ->
Int -> [(Int, Weight)] -> [(Int, Weight)]
floyd_warshall_in_place l_a l_b input b n k c_index big_replaced =
   if k == b then big_replaced
   else
       floyd_warshall_in_place l_a l_b input b n (k+1) c_index new_big_replaced
           where
               kth = k*n
               new_big_replaced = (loops k n kth 0 0 b l_a l_b input []
c_index)++big_replaced


inner_independent_phase :: Int -> Int -> Int -> Int -> Int -> [Weight] -> [(Int,
Weight)] -> [(Int, Weight)]
inner_independent_phase i j k b n input replaced =
   if j == (n `div` b) then replaced
   else
       if j == k then inner_independent_phase i (j+1) k b n input replaced
       else
           inner_independent_phase i (j+1) k b n input new_replaced
           where
               l_a = drop (i*b*n + k*b) input
               l_b = drop (k*b*n + j*b) input
               new_replaced = (floyd_warshall_in_place l_a l_b input b n 0 (i*b*n +
j*b) [])++replaced


independent_phase :: Int -> Int -> Int -> Int -> [Weight] -> [(Int, Weight)] ->
[Weight]
independent_phase i k b n input replaced =
   if i == (n `div` b) then res
   else
```

```
       if i == k then independent_phase (i+1) k b n input replaced
       else
           independent_phase (i+1) k b n new_input new_replaced2
               where
                   l_a = drop (i*b*n + k*b) input
                   l_b = drop (k*b*n + k*b) input
                   new_replaced = floyd_warshall_in_place l_a l_b input b n 0 (i*b*n +
k*b) []
                   new_input = replace_n_list new_replaced input
                   new_replaced2 = (inner_independent_phase i 0 k b n new_input
[])++replaced
                   res = replace_n_list replaced input


partially_dependent_phase :: Int -> [Weight] -> Int -> Int -> Int -> [(Int, Weight)]
partially_dependent_phase j input k n b =
   new_replaced
       where
           l_a = drop (k*b*n + k*b) input
           l_b = drop (k*b*n + j*b) input
           new_replaced = floyd_warshall_in_place l_a l_b input b n 0 (k*b*n + j*b) []

dependent_phase :: Int -> Int -> Int -> [Weight] -> [Weight]
dependent_phase k b n input =
   if k == (n `div` b) then input
   else
       dependent_phase (k + 1) b n new_in_output
           where
               l_a = drop (k*b*n + k*b) input
               l_b = drop (k*b*n + k*b) input
               big_replaced = floyd_warshall_in_place l_a l_b input b n 0 (k*b*n +
k*b) []
               new_dep_output = replace_n_list big_replaced input
               j_values = removeItem k [0..((n `div` b)-1)]
               big_replaced2 = concat (map (\j -> partially_dependent_phase j
new_dep_output k n b) j_values)
               new_part_output = replace_n_list big_replaced2 new_dep_output
               new_in_output = independent_phase 0 k b n new_part_output []

floyd_warshall_blocked :: [Weight] -> Int -> Int -> [Weight]
floyd_warshall_blocked input n b =
   dependent_phase 0 b n input
```

Parallel_fw_block.hs

```
{-# LANGUAGE DeriveAnyClass, DeriveGeneric #-}
{- Parallel Floyd-Warshall algorithm with 2-D block mapping in Haskell -}

module Parallel_fw_block (
       floyd_warshall_blocked
   ) where

import Graph
import Control.Parallel
import Control.Parallel.Strategies
import Control.Monad.Par(runPar, get, spawnP)
import Data.List
import Debug.Trace
import GHC.Generics (Generic)

loops :: Int -> Int -> Int -> Int -> Int -> Int -> [Weight] -> [Weight] -> [Weight] ->
[(Int, Weight)] -> Int -> [(Int, Weight)]
loops k n kth i j b l_a l_b input replaced c_index =
   if i == b then replaced
   else
       if j == b then loops k n kth (i + 1) 0 b l_a l_b input replaced c_index
       else
           if element > sum1 then loops k n kth i (j+1) b l_a l_b input new_replaced
c_index
           else
               loops k n kth i (j+1) b l_a l_b input replaced c_index
                   where element = dataAt (c_index + (i*n + j)) input
                         sum1 = addWeights (dataAt (i*n + k) l_a) (dataAt (kth + j)
l_b)
                         new_replaced = ((c_index + (i*n + j)), sum1):replaced


floyd_warshall_in_place :: [Weight] -> [Weight] -> [Weight] -> Int -> Int -> Int ->
Int -> [(Int, Weight)] -> [(Int, Weight)]
floyd_warshall_in_place l_a l_b input b n k c_index big_replaced =
   if k == b then big_replaced
   else
       floyd_warshall_in_place l_a l_b input b n (k+1) c_index new_big_replaced
```

```
           where
               kth = k*n
               new_big_replaced = (loops k n kth 0 0 b l_a l_b input []
c_index)++big_replaced


inner_independent_phase :: Int -> Int -> Int -> Int -> Int -> [Weight] -> [(Int,
Weight)]
inner_independent_phase j i k b n input =
   new_replaced
       where
           l_a = drop (i*b*n + k*b) input
           l_b = drop (k*b*n + j*b) input
           new_replaced = (floyd_warshall_in_place l_a l_b input b n 0 (i*b*n + j*b)
[])


independent_phase :: Int -> Int -> Int -> Int -> [Weight] -> [(Int, Weight)] ->
[Weight]
independent_phase i k b n input replaced =
   if i == (n `div` b) then res
   else
       if i == k then independent_phase (i+1) k b n input replaced
       else
           independent_phase (i+1) k b n new_input (new_replaced2++replaced)
               where
                   l_a = drop (i*b*n + k*b) input
                   l_b = drop (k*b*n + k*b) input
                   new_replaced = floyd_warshall_in_place l_a l_b input b n 0 (i*b*n +
k*b) []
                   new_input = replace_n_list new_replaced input
                   j_values = removeItem k [0..((n `div` b)-1)]
                   res = replace_n_list replaced input
                   new_replaced2 = runPar $ do
                       m <-  mapM (\j -> spawnP (inner_independent_phase j i k b n
new_input)) j_values
                       x <- mapM get m
                       return (concat x)

partially_dependent_phase :: Int -> [Weight] -> Int -> Int -> Int -> [(Int, Weight)]
partially_dependent_phase j input k n b =
   new_replaced
```

```
       where
           l_a = drop (k*b*n + k*b) input
           l_b = drop (k*b*n + j*b) input
           new_replaced = floyd_warshall_in_place l_a l_b input b n 0 (k*b*n + j*b) []

dependent_phase :: Int -> Int -> Int -> [Weight] -> [Weight]
dependent_phase k b n input =
   if k == (n `div` b) then input
   else
       dependent_phase (k + 1) b n new_in_output
           where
               l_a = drop (k*b*n + k*b) input
               l_b = drop (k*b*n + k*b) input
               big_replaced = floyd_warshall_in_place l_a l_b input b n 0 (k*b*n +
k*b) []
               new_dep_output = replace_n_list big_replaced input
               j_values = removeItem k [0..((n `div` b)-1)]
               new_part_output = replace_n_list big_replaced2 new_dep_output
               new_in_output = independent_phase 0 k b n new_part_output []
               big_replaced2 = runPar $ do
                   -- map (\j ...)  :: [Par (IVar [(,)])]
                   -- sequence (map ...) :: Par [IVar [(,)]]

                   m <-  mapM (\j -> spawnP (partially_dependent_phase j
new_dep_output k n b)) j_values
                   -- m :: [IVar [(,)]]

                   x <- mapM get m -- : [[(,)]]   -- mapM :: (a -> m b) -> [a] -> m
[b]
                   return (concat x)

floyd_warshall_blocked :: [Weight] -> Int -> Int -> [Weight]
floyd_warshall_blocked input n b =
   dependent_phase 0 b n input
```

Main.hs

```
{-# LANGUAGE DeriveAnyClass, DeriveGeneric #-}

import Graph
```

```haskell
import Parallel_fw_block
import Sequential_fw_block

import Control.Monad
import System.Random(randomIO, randomRIO)

import GHC.Generics (Generic)
import Control.Parallel
import Control.Parallel.Strategies

randomGraphGenerator :: Int -> Int -> Int -> [Weight] -> IO [Weight]
randomGraphGenerator num_of_vertices k i graph = do
   -- Every num_of_vertices * k + k is Weight 0
   -- otherwise assign a random weight or None
   -- when num_of_vertices == k - 1, return graph
   bool <- randomIO
   if i == (num_of_vertices*num_of_vertices) then do return (reverse graph)
   else do
       if ((num_of_vertices * k) + k) == i then do randomGraphGenerator
num_of_vertices (k+1) (i+1) (Weight 0:graph)
       else do
           if bool == False then do randomGraphGenerator num_of_vertices k (i+1)
(None:graph)
           else do
               let random_weight = randomRIO (-10, 200)
               w <- random_weight
               let new_weight = Weight w
               randomGraphGenerator num_of_vertices k (i+1) (new_weight:graph)

main :: IO ()
main = do
   g <- randomGraphGenerator 49 0 0 [] -- ::[Weight]
   --print g
   writeFile "file.txt" (show g)
   --print (Parallel_fw_block.floyd_warshall_blocked g 40 10)
   --print (Sequential_fw_block.floyd_warshall_blocked g 40 10)
```