# PolInc: Political Inclination in Social Networks

Sai Teja Reddy Moolamalla (sm5314)
Vikram Waradpande (vw2283)

## 1. Problem Statement

In large social networks, indicators of centrality can be used to assign rankings to people within a graph corresponding to their network position. Top influencers preaching for a particular political party directly influence their followers/first-level connections. Each person can directly preach their inclination to their contacts as well. Thus, we see a large-scale diffusion of information from each influencer to all the other nodes at a network level. We can score each person in the network based on the information they receive on each political party and use it to determine the overall sentiment of the population.

In this project, we attempt to simulate the flow of influence from top influencers in a social network to assign net scores to each member of the population which we plan to use to determine the **overall political inclination** of the population.

## 2. Problem Scope

To identify the political inclination of all the individuals in a social network, we will first identify the most influential people in the network using closeness centrality as a metric. Once we have a fixed number of such individuals, and we know their political inclination, we will simulate the spread of their influence (information) in the network using an information diffusion function. This function would be used to determine the effective influence of that information over a given person in the network. Once all the information has been propagated, we determine the **effective political inclination** of each person by summing up the effective scores from each influencer.

The scope of the project is to parallelize both the centrality computation as well as the diffusion. We will be using a few of the social networks from the **SNAP**[1] and **Network Repository**[2] datasets to run our simulations with various parameters.

## 3. Implementation and Algorithm Design

### a. Identification of top influencers in the network

To identify the top influencers in the social network we used the closeness centrality measures to rank all the nodes in the network and pick the top-ranked x% of nodes. **Closeness centrality** is a way of detecting nodes that are able to spread information very efficiently through a graph. The closeness centrality of a node measures its average farness (inverse distance) to all other nodes. Nodes with a high closeness score have the shortest distances to all other nodes.

For each node *u*, the Closeness Centrality algorithm calculates the sum of its distances to all other nodes, based on calculating the shortest paths between all pairs of nodes. The resulting sum is then inverted to determine the closeness centrality score for that node [3].

$$Closeness\ Centrality(u) \ = \ \frac{1}{\sum_{v:\ all\ nodes} shortestDistance(u, v)}$$

We relied on the *Floyd-Warshall* algorithm to find the shortest paths between each vertex pair in the graph. Once found, we computed the closeness centrality measures for all the nodes in the graph to identify the top influencers.

For larger networks of size greater than 35,000 nodes, both the serial and parallel versions' shortest path algorithm was not terminated within 120 mins, so we fell back to using the **degree centrality measure** to identify the top influencers for the rest of the algorithm.

## b. Simulate information diffusion from the influencers

Once the top influencers in the network are identified, we chose to cyclically assign labels and strengths proportional to the degree of each influencer. Then, we used an information diffusion function (I) to compute the influence of a node on its neighboring nodes. These neighboring nodes also are expected to spread the obtained information to their respective neighbors with an exponentially decreased strength. The below diffusion function which computes the influence of node ‘u’ on node ‘v’, was used for this simulation:

$$InfluenceU(v) = strength(u) * e^{-\alpha * distance(v, u)/diameter}$$

Thus, each influencer in the network diffuses some amount of information to every other node in the network. We implemented the Breadth-First Search (BFS) to compute the influence on all the nodes in the network due to an influencer, where the distance incrementally increases in the above function thereby reducing the influence exponentially for each BFS iteration.

The strength of the influencer is positive if the influencer favors the first political party and negative, if otherwise. Thus when the effective influence due to all influencers is computed and summed up for a particular node, we end up with the net score as below:

$$NetScore(v) = \sum_{U:\, influencers} InfluenceU(v)$$

If the net score for a node is positive, the inclination of the person is towards the first political party and towards the second party, if otherwise. Finally, votes for each party are computed based on the above inclinations to determine the overall sentiment of the population.

## 4. Datasets

We use two graph datasets to run our simulations.

1. Wiki Vote: The dataset contains all the Wikipedia voting data from the inception of Wikipedia till January 2008. Nodes in the network represent Wikipedia users and a directed edge from node ‘i’ to node ‘j’ represents that user ‘i’ voted on user ‘j’.

| | |
|---|---|
| Vertices | 889 |
| Edges | 2914 |

2. GitHub Social Network: This is a large social network of GitHub developers which was collected from the public API in June 2019. Nodes are developers who have starred at least 10 repositories and edges are mutual follower relationships between them.

| | |
|---|---|
| Vertices | 37,700 |
| Edges | 289,003 |

## 5. Performance

*Note: The performance below is based on running our code on an 8-core machine (Macbook Pro M1).*

To measure the parallelization efficiency of our algorithm we performed experiments on both graphs with a varying number of cores and a varying number of load chunks of the graph per core. First, we document the time taken by the sequential algorithm on both graphs.

## 5.1 Sequential Performance

We ran the sequential algorithm on both datasets. On the small graph, we used the closeness centrality measure and the degree measure on the larger graph to identify the top N% influencers.

Our observations are as follows:

- Average time taken by the sequential algorithm on **Graph 1** for the top 20% highest closeness centrality nodes was **234 seconds.**
- Average time taken by the sequential algorithm on **Graph 2** for the top 2% degree measure nodes was **172 seconds.**
- Average time taken by the sequential algorithm on **Graph 2** for the top 10% degree measure nodes was **956 seconds.**

## 5.2 Parallel Performance

There are two main components of the overall algorithm that we parallelized:

1. The identification of the influencer nodes
2. The simulation of spreading the influence function over the graph.

Furthermore, there are two ways to parallelize both these components, fixed chunking or dynamic chunking. In fixed chunking, each core gets a predefined workload, so there can be asymmetry in the total work performed by each worker. In the dynamic chunking approach, the units of work are distributed as evenly as possible. These both ways can be implemented in Haskell using the **parList** and **parListChunk** monads respectively. We performed experiments using both these monads. Our observations and findings are described below.

Experiments using parList and parMap: In our first attempt to parallelize our sequential algorithm we used parList and parMap to allow for parallel computations during list/map traversal. We ran our experiments with varying numbers of cores - 1/2/4/8/10/12 and recorded various attributes and compared the results.

**Experiment 1**: **parList** on Graph 1 with a varying number of cores.

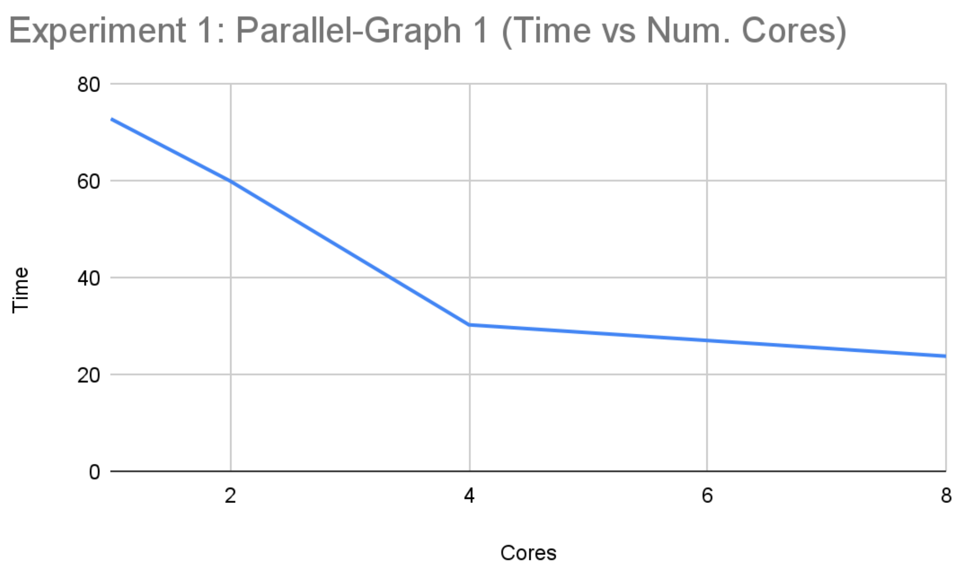


*Figure 1: Number of cores vs. Elapsed time in (secs) for Graph 1 (889 nodes)*

We observe that for the smaller graph (889 nodes), we see with an increase in the number of cores a significant speedup is achieved. The maximum speed-up is achieved for 8 cores which is 3.02 times the sequential version.

Below are the runtimes achieved using varying numbers of cores for the computations on graph 1:

| Number of Cores | Running Time (sec) | Speedup Achieved |
|---|---|---|
| 1 | 72.8 | 1 |
| 2 | 60.1 | 1.2 |
| 4 | 30.3 | 2.4 |
| 8 | 23.8 | **3.02** |

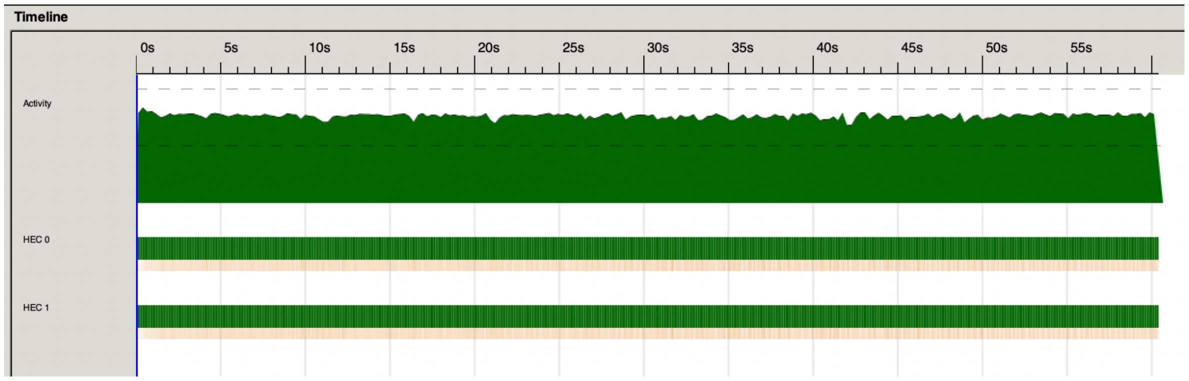


*Figure 2: Threadscope profile for the above experiment with 2 cores.*

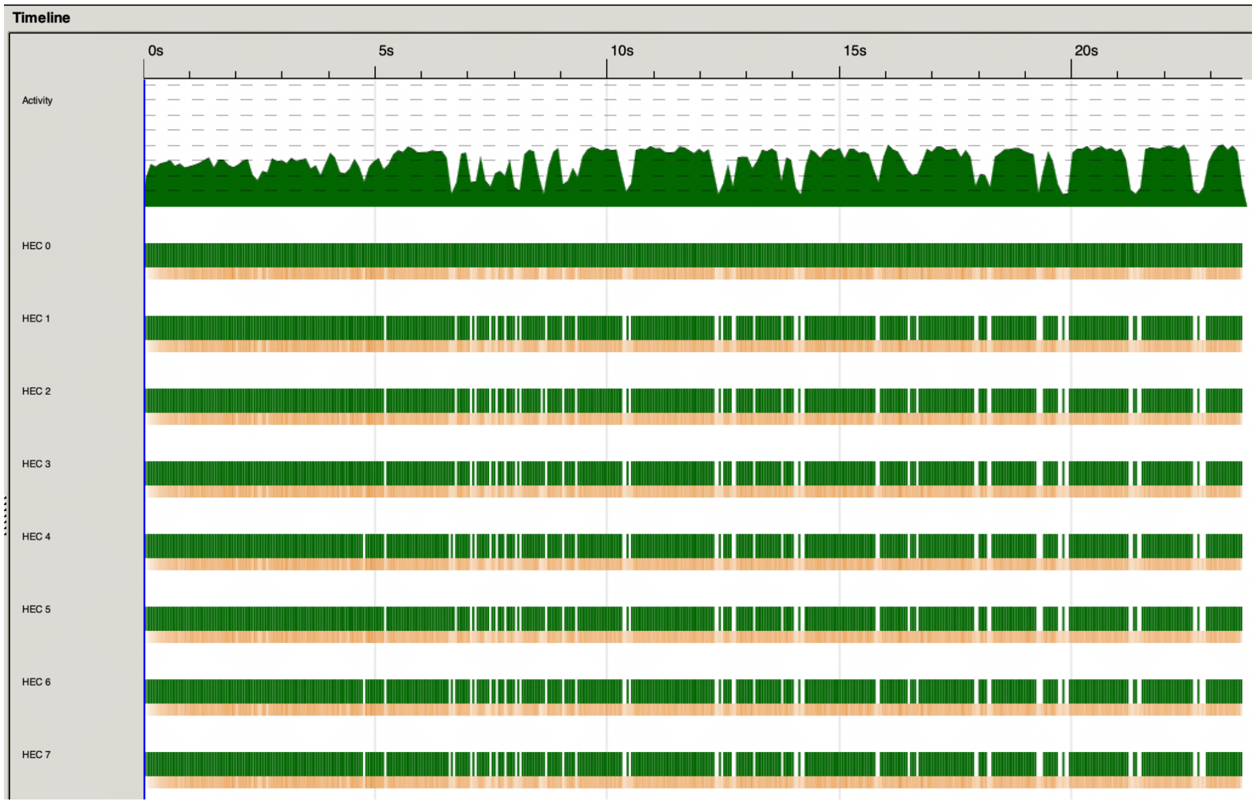


*Figure 3: Threadscope profile for the above experiment with 8 cores.*

Below is the spark profile achieved using 8-cores for graph 1:

```
 SPARKS: 791388 (730901 converted, 0 overflowed, 0 dud, 56012 GC'd, 4475 fizzled)

  INIT    time    0.000s  (  0.004s elapsed)
```

```
MUT     time   69.775s  ( 13.338s elapsed)
GC      time   32.521s  ( 10.302s elapsed)
EXIT    time    0.001s  (  0.011s elapsed)
Total   time  102.296s  ( 23.655s elapsed)
```

**Experiment 2**: **parList** on Graph 2 with top 2% influencers (based on degree measures)

We attempted running the same parallel algorithm on Graph 2 with closeness centrality measures but the algorithm wouldn't terminate for over 2 hrs, so we relied on faster-computed degree measures to identify the influencer nodes.

We observe that for the larger graph (37,000 nodes), we see with an increase in the number of cores a significant speedup is achieved. The maximum speed-up is achieved for 8 cores which is **2.89** times the sequential version.

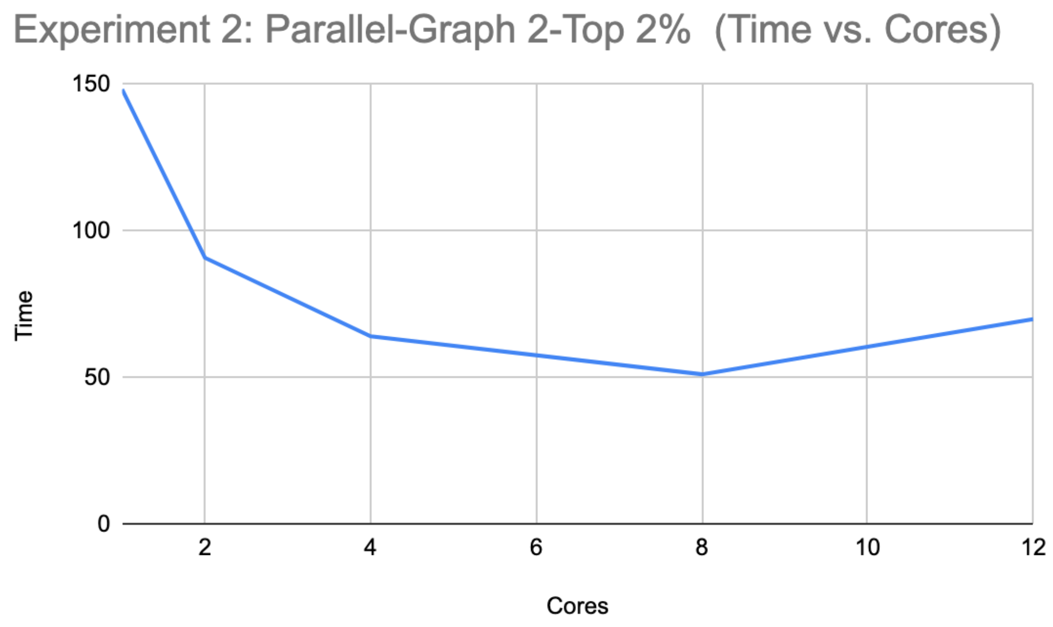


*Figure 4: Number of cores vs. Elapsed time in (secs) for Graph 2 - Top 2% (37,700 nodes)*

Below are the runtimes achieved using varying numbers of cores for the computations on graph 2:

| Number of Cores | Running Time (sec) | Speedup Achieved |
|---|---|---|
| 1 | 148.24 | 1 |

| | | |
|---|---|---|
| 2 | 90.84 | 1.63 |
| 4 | 64.1 | 2.31 |
| 8 | 51.13 | **2.89** |
| 12 | 69.93 | 2.11 |

An interesting observation is that using 12-cores is actually slower than using 8-cores. This could be attributed to the increased overhead of communicating across more cores.

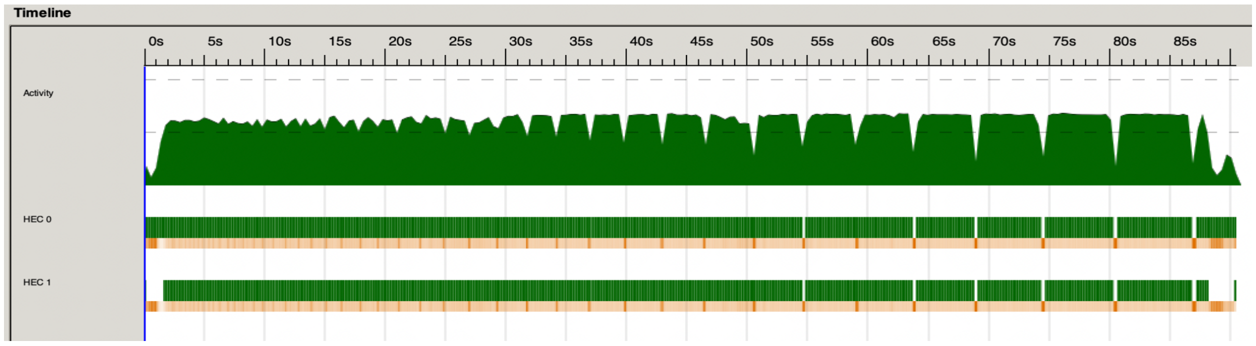


*Figure 5: Threadscope for the above experiment with 2 cores.*

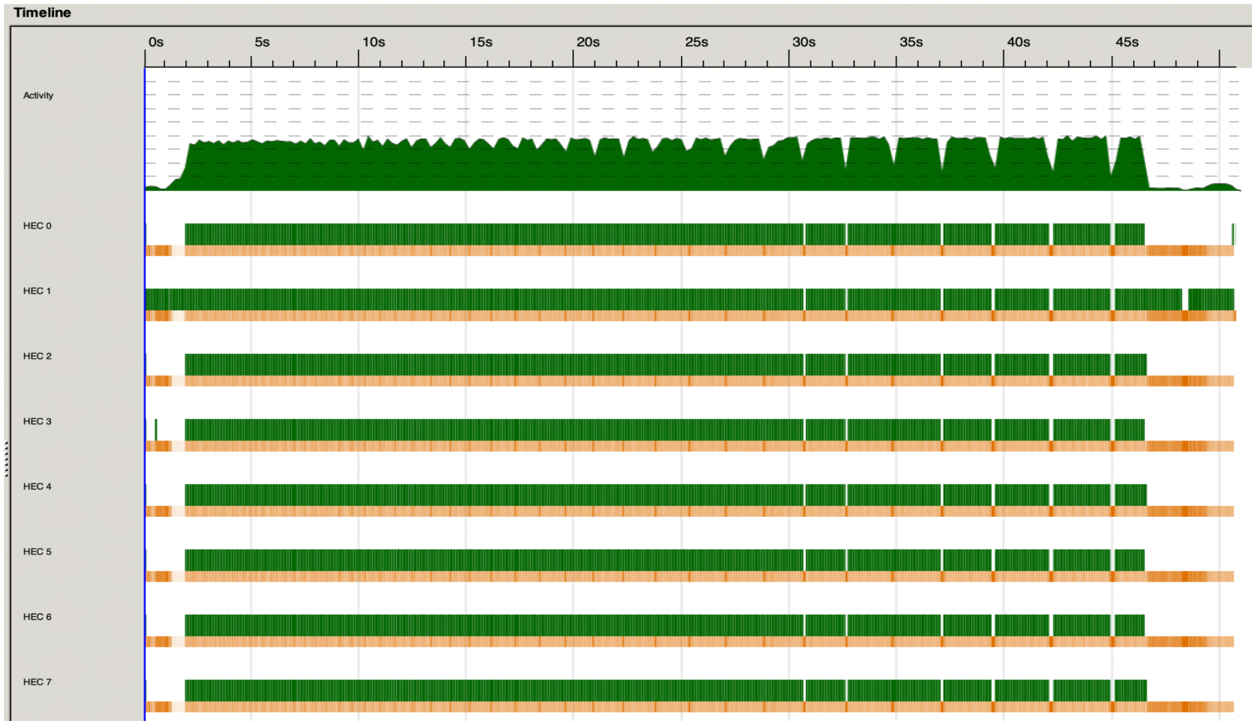


*Figure 6: Threadscope for the above experiment with 8 cores.*

**Experiment 3**: **parList** on Graph 2 with top 10% influencers (based on degree measures)

We ran an experiment on Graph 2 with the top 10% influencers to observe how the algorithm scales with more work. Interestingly, we've observed a lower speedup of **1.98**, even though the peak was using 8 cores.

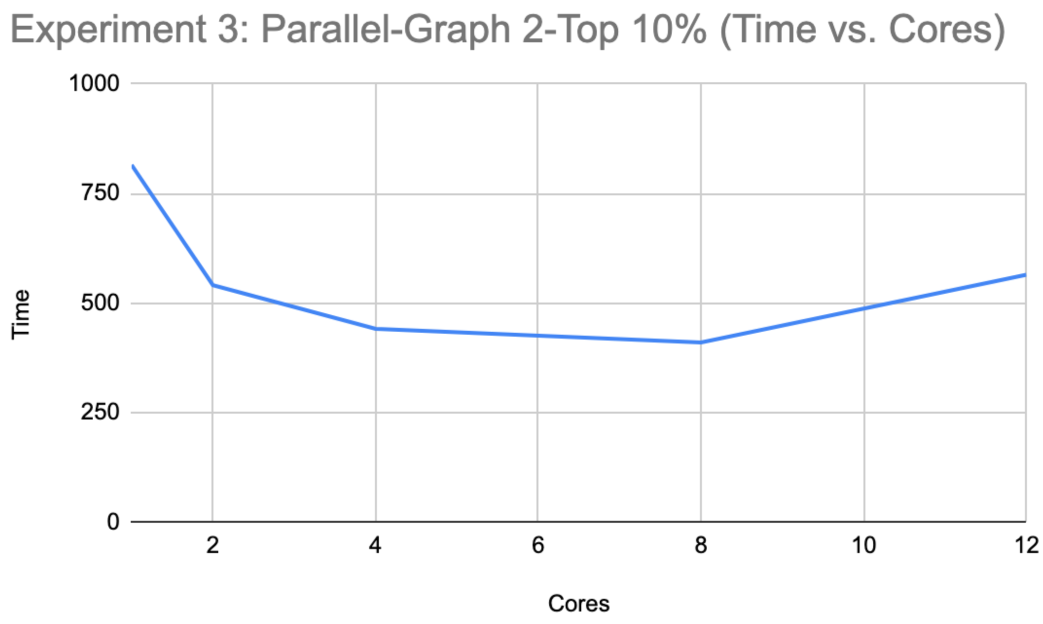
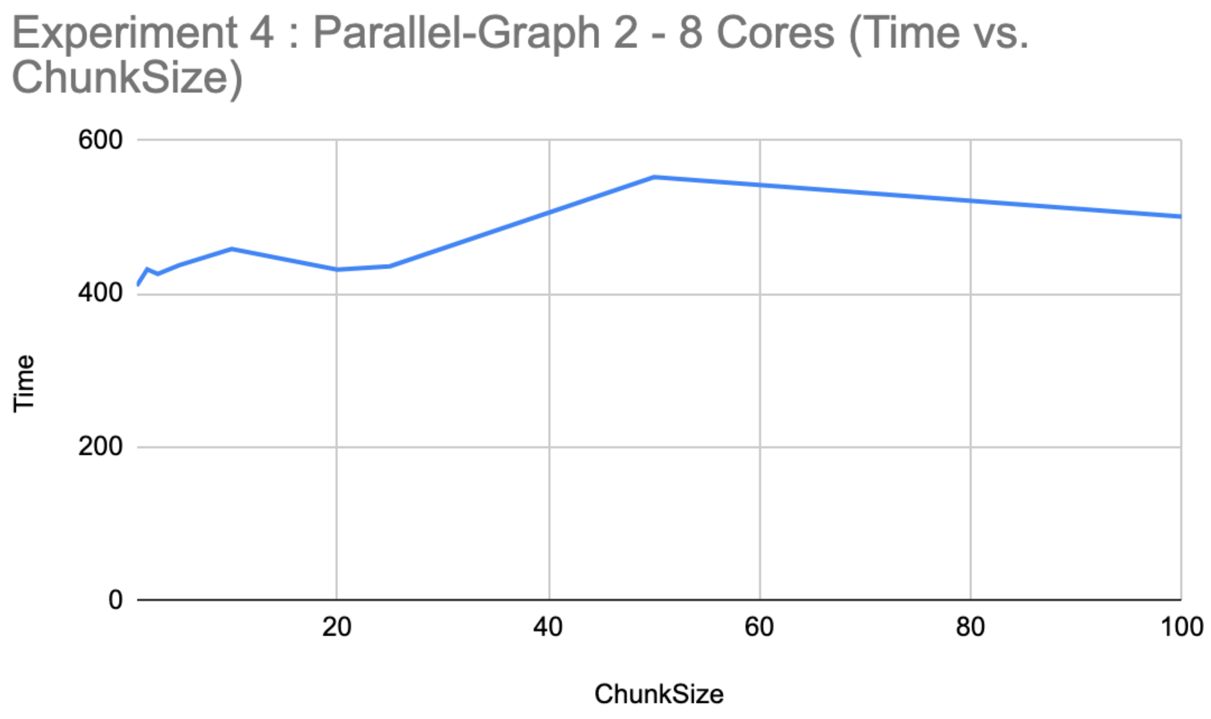


*Figure 7: Number of cores vs. Elapsed time in (secs) for Graph 2 - Top 10% (37,700 nodes)*

| Number of Cores | Running Time (sec) | Speedup Achieved |
|---|---|---|
| 1 | 815.1 | 1 |
| 2 | 540.64 | 1.50 |
| 4 | 441.0 | 1.84 |
| 8 | 410.14 | **1.98** |
| 12 | 564.3 | 1.44 |

**Experiment 4: parListChunk** on Graph 2, running on 8 cores, 2% influencers and varying chunk size.

Here chunkSize is defined as the number of vertices that each core processes for simulating the spreading influence throughout the graph.



| Chunk Size | Running Time (sec) | Sparks (converted) | Speedup (compared to sequential) |
|---|---|---|---|
| 1 | 410.1 | 3770 (3348) | **1.98** |
| 2 | 431.7 | 1885 (1884) | 1.88 |
| 3 | 425.83 | 1257 (1256) | 1.91 |
| 5 | 437.06 | 754 (753) | 1.86 |
| 10 | 458.4 | 377 (376) | 1.77 |
| 20 | 431.5 | 189 (188) | 1.88 |
| 25 | 435.8 | 151 (150) | 1.87 |
| 50 | 436.6 | 76 (75) | 1.86 |
| 100 | 453.5 | 38 (37) | 1.78 |

Note that Chunk Size = 1 corresponds to the earlier experiment, equivalent to **parList** as at each point a processor is assigned just one vertex/unit of work. As the chunk size increases, the speedup shows an interesting trend, where increasing the chunkSize decreases the performance. We observe the maximum speedup with a chunk size of 1 where all the influencer nodes are evenly distributed across each spark.

## 6. Conclusion

In conclusion, we achieved a maximum speedup of **3.02** on the smaller graph, and a maximum speedup of **1.98** on the larger graph for 8 cores. We explored two different ways to parallelize our algorithm and did a comparative analysis of both.

## 7. Future Enhancements

One of the inferences from experiments 3 and 4 is that dividing the total work equally actually saves a lot of time in this problem. For instance when we compare chunk size = 1 and chunk size = 5, the former leads to a more equal distribution of workload across the workers. However, in our current implementation, there is scope for fine-graining even more equitable work distribution in the shortest paths approach. Our current parallel workflow can be thought as distributing each node to a worker and performing a complete BFS/Closeness measure on that worker for that node. However, each exploration of a vertex during the BFS can also be parallelized on larger graphs, which can lead to even better performance.

Another enhancement would be to use more sophisticated all-pairs-shortest-path algorithms [4] for better performance.

The algorithm can be adapted to help perform some experiments like assigning a different number of influencers to different parties (rather than equal), using a more complex influence function, using another centrality metric to identify influential individuals, etc. We can also examine how many relatively weak influencers of a political party it takes to outweigh fewer but more influential influencers elsewhere in the network. This can offer more insights into the structure of the network.

## 8. References

1. Stanford Network Analysis Project
2. Network Respository - Social Network Graphs
3. Closeness Centrality - Neo4j Graph Data Science
4. Distributed Algorithms for All Pairs Shortest Paths

## 9. Code Listing

Main.hs

```haskell
module Main (main) where

import Lib
import System.Environment
import System.Exit

main :: IO ()
main = do
       args <- getArgs
       case args of
           [parSeq, chunkSize ,fileName] -> do
               putStrLn parSeq
               putStrLn $ show chunkSize
               case parSeq of
                   "par" -> do
                       polIncParChunk (read chunkSize :: Int) fileName
                   _ ->
                       polIncSerial fileName
           [parSeq, fileName] -> do
               putStrLn parSeq
               case parSeq of
                   "par" -> do
                       polIncPar fileName
                   _ ->
                       polIncSerial fileName
           _ -> do
```

```
              die "exit"
```

Lib.hs

```
module Lib
   ( polIncPar,
     polIncParChunk,
     ppolIncSerial
   ) where

import qualified Data.Vector as V
import Data.List
import Control.Parallel.Strategies

import MapCompat (IntMap)
import qualified Data.IntMap as Map
import Data.Ord (comparing)
import qualified Data.Set as S
import qualified Control.Monad.State as StM
import Data.List (sortBy, group, sort)
import Control.Arrow ((&&&))
import qualified Data.List.Split as Split
import qualified Data.ByteString.Lazy as BL
import qualified Data.Csv as CSV
import qualified Data.IntSet as IntSet

type Vertex = Int
type Weight = Int
type Score = Double
type Graph = IntMap (IntMap Weight)

--Functions to read the edgelist, build and return a graph

weight :: Graph -> Vertex -> Vertex -> Maybe Weight
weight g i j = do
 jmap <- Map.lookup i g
 Map.lookup j jmap

insertEdge :: Vertex -> Vertex -> Weight -> Graph -> Graph
insertEdge i j w m = Map.insert i iarr m'
         where iarr = Map.insert j w ia
               ia = Map.findWithDefault Map.empty i m'
               m' = Map.insert j jarr m
               jarr = Map.insert i w ja
```

```haskell
               ja = Map.findWithDefault Map.empty j m


readGraph :: String -> IO (V.Vector (Int, Int))
readGraph fileName = do
   let f = fileName
   BL.readFile f >>= dat . (CSV.decode CSV.NoHeader)
   where dat (Left _) = undefined
         dat (Right v) = return $ app v
           where app v' = do
                           (v1 :: Int, v2 :: Int) <- v'
                           return (v1, v2)

getGraph :: [(Vertex, Vertex)] -> (Graph, [Vertex])
getGraph edges = (mat, vers)
   where
       vers = getVertices edges
       mat = foldr ins Map.empty edges
           where ins (i,j) = insertEdge i j 1

getVertices' :: [(Vertex, Vertex)] -> [Vertex]
getVertices' [] = []
getVertices' ((v1, v2):xs) = [v1, v2] ++ (getVertices' xs)

getVertices :: [(Vertex, Vertex)] -> [Vertex]
getVertices edges = IntSet.elems (IntSet.fromList $ getVertices' edges)


--Function to return the top N% of the vertices in the list
topPercent :: Double -> Int -> Int
topPercent percent count | ((fromIntegral count)*percent/100.0) > 1.0 = fromIntegral $
ceiling $ (fromIntegral count)*percent/100.0
                | otherwise = 1

--Function to find the shortest path from a given vertex to everyvertex in the graph
processVertexMap :: Vertex -> [Vertex] -> Graph -> Vertex -> IntMap Weight -> IntMap
Weight
processVertexMap k vs g i jmap = foldr shortest Map.empty vs
   where shortest j m =
           case (old,new) of
             (Nothing, Nothing) -> m
             (Nothing, Just w ) -> Map.insert j w m
             (Just w,  Nothing) -> Map.insert j w m
             (Just w1, Just w2) -> Map.insert j (min w1 w2) m
           where
             old = Map.lookup j jmap
             new = do
```

```
                     if (i == j)
                       then do return 0
                     else do
                           w1 <- weight g i k
                           w2 <- weight g k j
                           return (w1+w2)

--Parallel implementation of shortest path for each vertex in the graph
shortestPathsParallel :: [Vertex] -> Graph -> Graph
shortestPathsParallel vs g = foldl' update g vs
where
 update g k = Map.fromList $ parMap rdeepseq (\(i, jmap) -> (i, (processVertexMap k vs
g) i jmap)) (Map.toList g)


--Sequential implementation of shortest path for each vertex in the graph
shortestPaths :: [Vertex] -> Graph -> Graph
shortestPaths vs g = foldl' update g vs
where
 update g k = Map.mapWithKey (processVertexMap k vs g) g


degreeMeasures :: Graph -> [(Int, Int)]
degreeMeasures graph = [ (a, length b) | (a, b) <- (Map.toList graph)]

closenessMeasures :: Graph -> [(Int, Int)]
closenessMeasures shortestPaths = [ (a, sum $ Map.elems b) | (a, b) <- (Map.toList
shortestPaths)]

closenessMeasuresHelper :: Graph -> [(Int, Int)]
closenessMeasuresHelper sp = concat (map (closenessMeasures.(Map.fromList)) chunks
`using` parList rdeepseq)
                    where chunks = Split.chunksOf (length sp `quot` 8) (Map.toList sp)

closenessMeasuresHelperNoChunks :: Graph -> [(Int, Int)]
closenessMeasuresHelperNoChunks shortestPaths = concat (map
(closenessMeasures.Map.fromList.(\x -> [x])) (Map.toList shortestPaths) `using`
parList rdeepseq)

countOccurences :: Ord a => [a] -> [(a, Int)]
countOccurences = map (head &&& length) . group . sort

getCounts :: [(Int, Score)] -> [(Int, Int)]
getCounts scores = countOccurences.Map.elems.Map.fromList $ map (\(a, b) -> (a, (\c ->
if c >= 0 then 1 else -1) b)) scores
```

```haskell
--Calculate summary of how many nodes of each of each political party/inclication
there are
computeSummary :: IntMap Score -> IO ()
computeSummary scores = do
              mapM_ (putStrLn.show) (getCounts $ Map.toList scores)

--Look up the adj list to see which vertices are adjacent to given vertex
getNeighbours ::  Int -> Graph -> [Vertex]
getNeighbours source graph = case (Map.lookup source graph) of
                                Nothing -> []
                                Just v -> Map.keys v

--Using the equation given in the report, calculate the 'strenth' to be assigned
diffuse :: Double -> Double -> Double -> Double -> Double
diffuse strength distance diameter alpha = strength * (exp $ (-1.0) * alpha *
distance/diameter)

--BFS Helper to identify fringe
getNextSources :: Int -> [Vertex] -> Graph -> Double -> StM.State (IntMap Score)
[Vertex]
getNextSources dist sources graph strength = do
   visited <- StM.get
   let newNeigh = S.fromList $ concat [getNeighbours s graph | s <- sources]
       nextsources = S.toList $ S.difference newNeigh $ S.fromList $ Map.keys visited
       score = diffuse strength (fromIntegral dist) 6 1
   StM.put $ foldr (\a -> Map.insert a score) visited nextsources
   return nextsources

--standard BFS
doBFS :: Int -> [Vertex] -> Graph -> Double -> StM.State (IntMap Score) (Graph)
doBFS dist sources graph strength = do
     newsources <- getNextSources dist sources graph strength
     if | length newsources == 0 -> return graph
        | otherwise -> doBFS (dist + 1) newsources graph strength

generateScores:: Graph -> (Vertex, Double) -> IntMap Score
generateScores graph (source, strength) = StM.execState (doBFS 1 [source] graph
strength) (Map.fromList [(source, 0)])

getScores:: Graph -> [(Vertex, Double)] -> IntMap Score
getScores graph influencers = foldr (\a b -> Map.unionWith (+) (generateScores graph
a) b) Map.empty influencers

getScoresHelper:: Graph -> [(Vertex, Double)] -> Int -> IntMap Score
```

```
getScoresHelper graph influencers chunkSize = foldr (\a b -> Map.unionWith (+) a b)
Map.empty (map (generateScores graph) influencers `using` parListChunk chunkSize
rdeepseq)


getScoresHelperNoChunks:: Graph -> [(Vertex, Double)] -> IntMap Score
getScoresHelperNoChunks graph influencers = foldr (\a b -> Map.unionWith (+) a b)
Map.empty (map ((getScores graph).(\x -> [x])) influencers `using` parList rdeepseq)


--Entry point if supplied arguments have a "par" and a chunkSize (and filename)
polIncParChunk :: Int -> String -> IO ()
polIncParChunk chunkSize fileName = do
   graph <- readGraph fileName
   let edges = V.toList graph
       (mat,vs) = getGraph edges
       -- sp = shortestPathsParallel vs mat
       deg = degreeMeasures mat
       -- cm = closenessMeasuresHelper sp
       -- influencers = map fst $ take (topPercent 20 $ length vs) $ sortBy (comparing
snd) $ cm
       influencers = map fst $ take (topPercent 10 $ length vs) $ reverse $ sortBy
(comparing snd) $ deg
   putStrLn $ show $ length vs
   putStrLn "polIncParChunk running"
   let netscores = getScoresHelper mat (zip influencers (cycle [1, -1])) chunkSize
   computeSummary netscores


--Entry point if supplied arguments have just a "par" (and filename)
polIncPar :: String -> IO ()
polIncPar fileName = do
   graph <- readGraph fileName
   let edges = V.toList graph
       (mat,vs) = getGraph edges
       -- sp = shortestPathsParallel vs mat
       deg = degreeMeasures mat
       -- cm = closenessMeasuresHelperNoChunks sp
       -- influencers = map fst $ take (topPercent 20 $ length vs) $ sortBy (comparing
snd) $ cm
       influencers = map fst $ take (topPercent 10 $ length vs) $ reverse $ sortBy
(comparing snd) $ deg
   putStrLn $ show $ length vs
   putStrLn "polIncPar running"
   let netscores = getScoresHelperNoChunks mat (zip influencers (cycle [1, -1]))
   computeSummary netscores
```

```haskell
----Entry point for sequential version
polIncSerial :: String -> IO ()
polIncSerial fileName = do
   graph <- readGraph fileName
   let edges = V.toList graph
       (mat,vs) = getGraph edges
       -- sp = shortestPaths vs mat
       deg = degreeMeasures mat
       -- cm = closenessMeasures sp
       -- influencers = map fst $ take (topPercent 20 $ length vs) $ sortBy (comparing
snd) $ cm
       influencers = map fst $ take (topPercent 10 $ length vs) $ reverse $ sortBy
(comparing snd) $ deg
   putStrLn $ show $ length vs
   putStrLn "polIncSerial running"
   let netscores = getScores mat (zip influencers (cycle [1, -1]))
   computeSummary netscores
```

Sample Output:

```
(base) saiteja:pollinc/ (main✗) $ time stack exec pollinc-exe "par" 100
../git_web_ml/musae_git_edges.csv  -- +RTS -N8 -l -s
par
"100"
37700
polIncParChunk running
(-1,19385)
(1,18315)
```

Since, party -1 has more votes, the population leans towards this party!