Project Report: Optimizing Financial Contagion Modeling

Erica Choi (eyc2130), Nick Ching (nc2935)

November 17, 2024

View our GitHub repository

Abstract

The propagation of financial shocks across interconnected systems represents a critical challenge in understanding and managing systemic risk. Accordingly, Influence Maximization (IM) problems offer a framework for modeling the diffusion of financial shocks in a network. However, the computational expense of existing methods, often reliant on sequential Monte Carlo simulations limit the scale and scope of IM models. This report presents a parallelized implementation of the greedy algorithm for the Independent Cascade Model (ICM). Utilizing Haskell, our results will demonstrate that parallelization accomplishes significant reductions in computation time, enabling more complex analysis of financial shock propagation. This work showcases the effectiveness of parallelism in computational finance.

1 Background

The study of cascading failures in networks, such as financial contagion, draws on models like the Independent Cascade Model (ICM), a probabilistic framework widely used in influence maximization and diffusion studies. In financial systems, entities can propagate shocks to their neighbors, potentially leading to widespread failures. Identifying the degree of influence propagation that certain institutions have is useful in systemic risk analysis.

The influence maximization problem, introduced by Domingos and Richardson [1], seeks to identify a set of nodes in a network that maximizes the spread of influence under models like ICM. While greedy algorithms have been proven to provide near-optimal solutions with provable guarantees, their reliance on Monte Carlo simulations makes them computationally expensive for large graphs.

Haskell, with its concurrency model and parallel computation features, offers a promising platform for implementing scalable solutions. This project develops a scalable tool that, given a set of institutions of interest as an input, metricizes the expected effect of financial shock in the network.

2 Problem Formulation

The problem is formalized as follows: given a financial network represented as a directed graph $G = (V, E)$ where V denotes the set of entities (nodes) and E is the set of relationships (edges), each edge $e \in E$ is assigned a weight $w(e) \in [0.1, 0.5]$ representing the probability of influence propagation.
Given a seed set $S \subseteq V$, the Independent Cascade Model (ICM) simulates the probabilistic diffusion of influence as follows:

1. Initially, all nodes in S are active, and all other nodes are inactive.
2. At each iteration, active nodes attempt to activate their inactive neighbors with a probability defined by the edge weight.
3. The process continues until no new nodes are activated.

The objective is to compute the expected number of influenced nodes in V given a subset (the seed set) $S \subseteq V$. In this report, we focus on the case when $|S| = 1$, which can be easily generalized to the case where $|S| > 1$. Due to the stochastic nature of the ICM, Monte Carlo simulations are typically used, which is computationally expensive.
This project thus focuses on optimizing the ICM's computation using parallelism to improve efficiency and scalability.

3 Methodology

Development of our program evolved over three phases, each building on insights from the previous.

3.1 Phase 1: Sequential Greedy Implementation

As a baseline, we implemented a sequential greedy algorithm for the ICM. This version iteratively computes the influence spread of a given seed set S using Monte Carlo simulations. Each simulation follows the probabilistic activation rules of the ICM, with random number generation driving the diffusion process. The sequential approach demonstrated significant computational overhead, particularly for larger graphs and high numbers of simulations.

3.2 Phase 2: Naïve Parallelization

In the second phase, we attempted to parallelize the Monte Carlo simulations by dividing the total number of simulations across multiple cores. While the algorithm successfully distributed the simulations, the implementation relied on a shared random number generator, introducing catastrophic inefficiencies. As a result, the naïve parallelized version exhibited slower performance than the sequential greedy implementation. This highlighted the importance of managing random number generation in parallelized stochastic simulations.

3.3 Phase 3: Fully Parallelized Implementation

In the final phase, we developed a fully parallelized version of the ICM that addressed the inefficiencies in the random number generation. By utilizing `getStdGen`, which allows random number generators to be split and assigned to each parallel thread, we ensured that simulations ran concurrently without interference. This corrected implementation leveraged Haskell's `Control.Parallel.Strategies` to divide simulations into chunks distributed across multiple cores.

3.4 Benchmarking and Analysis

Each implementation was benchmarked on graphs with 30,000 nodes and 300,000 weighted edges, running 1,000 Monte Carlo simulations for each graph. Performance metrics include:

- **Execution Time:** Measured for each implementation to evaluate computational efficiency.
- **Scalability:** Assessed by running the algorithms on graphs of increasing size and complexity.
- **Correctness:** Verified by comparing the influence spread results across all implementations.

4 Implementation

The following algorithms in pseudocode illustrate our implementation of parallelization strategies for the Independent Cascade Model (ICM). This section focuses on the ICM execution rather than the preliminary steps required to construct the underlying graph structures. For details regarding the general procedures used to build each graph, please refer to Appendix A.

Algorithm 1 Naïve ICM

Inputs: - Graph $G = (V, E)$ with edge probabilities w_{uv}, - Seed set S, - Number of simulations numSim,

1: **function** INDEPENDENTCASCADE(G, S):
2: $\quad$ $activated \leftarrow S$, $current \leftarrow S$
3: $\quad$ **while** $current \neq \emptyset$ **do**:
4: $\quad\quad$ $newlyActivated \leftarrow \emptyset$
5: $\quad\quad$ **for** $u \in current$ **do**:
6: $\quad\quad\quad$ **for** $(v, w) \in \text{neighbors}(u, G)$ **do**:
7: $\quad\quad\quad\quad$ **if** $v \notin activated$ and $\text{random}(0, 1) \leq w$ **then**:
8: $\quad\quad\quad\quad\quad$ $newlyActivated \leftarrow newlyActivated \cup \{v\}$
9: $\quad\quad\quad\quad$ **end if**
10: $\quad\quad\quad$ **end for**
11: $\quad\quad$ **end for**
12: $\quad\quad$ $activated \leftarrow activated \cup newlyActivated$, $current \leftarrow newlyActivated$
13: $\quad$ **end while**
14: $\quad$ **return** $activated$
15: **end function**
16: **function** MONTECARLOSIMULATION(G, S, numSim):
17: $\quad$ **return** $\frac{1}{\text{numSim}} \sum_{i=1}^{\text{numSim}} |\text{INDEPENDENTCASCADE}(G, S)|$
18: **end function**

Algorithm 2 Parallelized ICM

Inputs: - Graph $G = (V, E)$ with edge probabilities w_{uv}, - Seed set S, - Number of simulations numSim, - Available cores: getNumCapabilities(),

```
1: function IndependentCascade(G, S):
2:     activated ← S, current ← S
3:     while current ≠ ∅ do:
4:         newlyActivated ← ∅
5:         for u ∈ current do:
6:             for (v, w) ∈ neighbors(u, G) do:
7:                 if v ∉ activated and random(0, 1) ≤ w then:
8:                     newlyActivated ← newlyActivated ∪ {v}
9:                 end if
10:            end for
11:        end for
12:        activated ← activated ∪ newlyActivated, current ← newlyActivated
13:    end while
14:    return |activated|
15: end function
16: function MonteCarloSimulation(G, S, numSim):
17:    cores ← getNumCapabilities()
18:    gens ← take(numSims, iterate(λg : snd(split(g)), stdGen))
19:    chunkSize ← ⌈numSim/cores⌉
20:    chunks ← chunkList([1, . . . , numSim], chunkSize)
21:    partialResults ← parMap(λc : simulateChunk(G, S, c), chunks)
22:    return ∑(partialResults)/numSim
23: end function
24: function simulateChunk(G, S, chunk):
25:    result ← 0
26:    for i ∈ chunk do:
27:        result ← result + IndependentCascade(G, S)
28:    end for
29:    return result
30: end function
31: function chunkList(list, size):
32:    return Divide list into sublists of at most size elements
33: end function
```

5 Parallelization Techniques

Our project utilizes Haskell's parallelization primitives to execute the ICM monte carlo simulations across multiple cores. In doing so, we are able to accomplish significant speedup.

5.1 Chunking Simulations

Rather than sequentially executing each simulation, the `monteCarloSimulation` function chunks the total number of simulations to be executed on separate CPU cores in parallel. The `getNumCapabilities` function will return the number of cores available and, by extension, how many chunks to make.

5.2 parMap and rdeepseq for Parallel Evaluation

Utilizing `parMap` and `rdeepseq`, we are able to evaluate each chunk in parallel. In particular, `parMap` allows us to compute parallel functions over a set list, and `rdeepseq` ensures that the computations are fully evaluated. This guarantees that we will not be considering lazy thunks before aggregating the final result.
In particular, we do get a benefit to using parMap. In our case, parMap will return a list of partial computations representing each chunk's summation of propagation. Only after each chunk has been computed do we sum the list of partial computations. This means that the vast majority of additions are done in parallel, improving computation time.

5.3 Random number generation

An initial problem we faced with a naïve implementation of parallelization is the random number generation. In our initial implementation that used `randomRIO`, the runtime of the simulations actually proved to be significantly worse than the sequential implementation's.

This slowdown resulted from multiple threads attempting to use a single global random number generator that resulted in conflicts. We addressed this issue by obtaining the global standard generator, `StdGen`, from `getStdGen`. Subsequently, using the `split` function to create a list of independent random number generators.

Each simulation receives a distinct `StdGen` from the generated list of random number generators. This optimization kills two birds with one stone. It firstly avoids unnecessary resource contention, but also ensures statistical independence for each simulation thereby guaranteeing correctness

5.4 Impact of parallelization techniques

By chunking our simulations and simultaneously processing each chunk, idle CPU time is significantly decreased. Given that each simulation is independent, and the independent nature of Monte Carlo simulations, our code is able to effectively reduce computation time with an increase in computing power.

6 Results

6.1 Summary

The following table represents the conclusions of our progressive parallelization of the ICM model. At first glance, it becomes immediately apparent that the naïve parallel implementation of the ICM model showed significant slowdowns. This reflects the incorrect usage of random number generators, which acted as a bottle-neck for each thread.

Method	Time to Complete (s)
Sequential Implementation	434.379
Naïve Parallel ICM	803.318
Parallel ICM	70.726

Table 1: Comparison of three approaches to the ICM simulation.

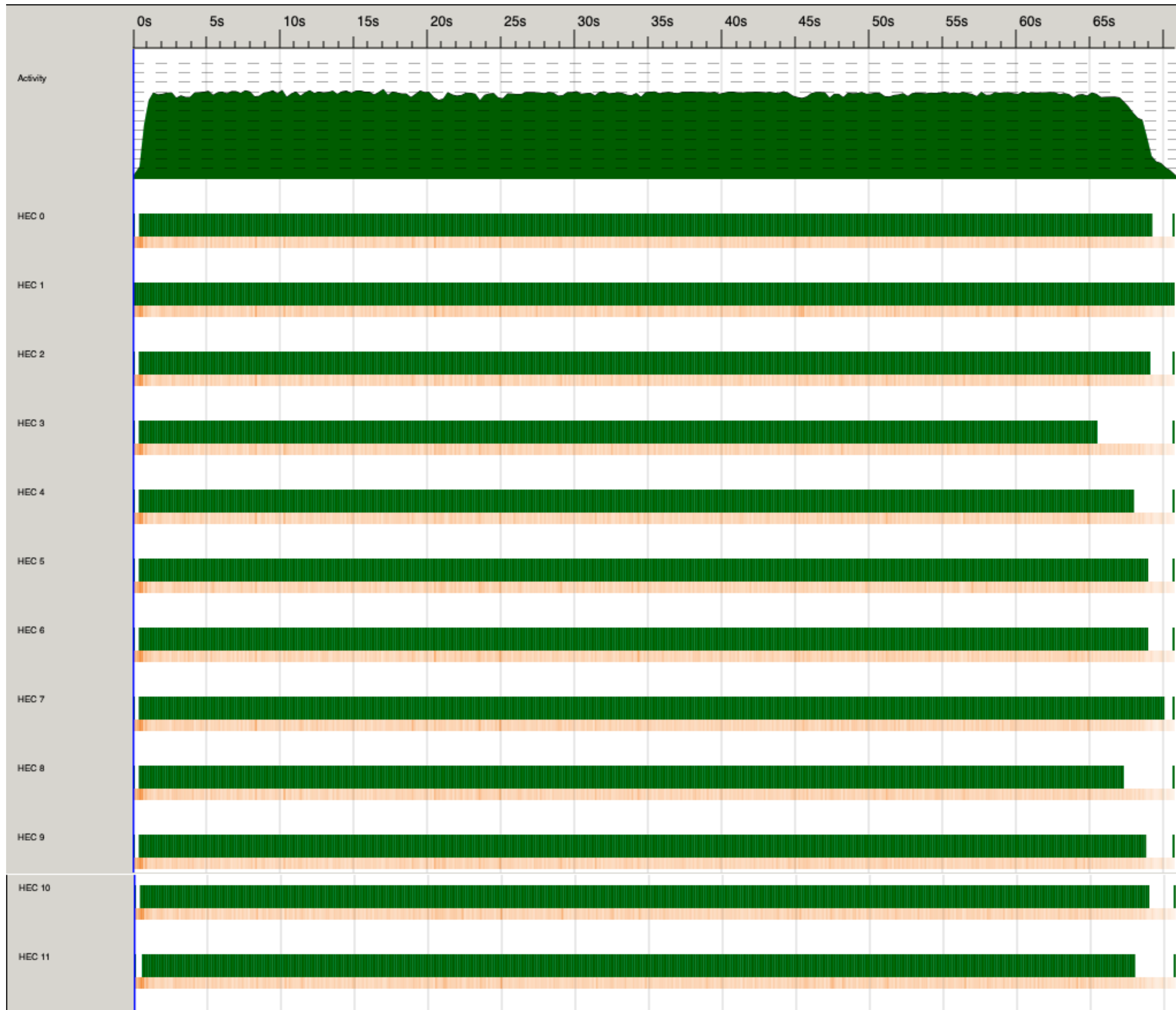
However, after resolving the bottleneck our optimized solution accomplished a **6.14x** speedup. And optimized parallelization achieves an **11.36x** speedup over the naïve parallel ICM implementation.

6.2 Optimized Parallel ICM Sparks

```
~/De/Col/1. Parallel Functional Programming/shocknet main !2 ?13 ❯ time ./parallel_icm +RTS -N12 -ls -s
Building graph...
Graph built. Running Monte Carlo simulations...
Average Influence (Monte Carlo): 26574.094
 638,731,222,504 bytes allocated in the heap
  47,954,528,320 bytes copied during GC
      77,935,424 bytes maximum residency (274 sample(s))
       1,292,936 bytes maximum slop
             285 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     19870 colls, 19870 par   56.524s  10.998s     0.0006s    0.0055s
  Gen  1       274 colls,   273 par   20.966s   2.593s     0.0095s    0.0281s

  Parallel GC work balance: 75.98% (serial 0%, perfect 100%)

  TASKS: 26 (1 bound, 25 peak workers (25 total), using -N12)

  SPARKS: 12 (11 converted, 0 overflowed, 0 dud, 0 GC'd, 1 fizzled)

  INIT    time    0.010s  (  0.010s elapsed)
  MUT     time  551.244s  ( 57.121s elapsed)
  GC      time   77.489s  ( 13.591s elapsed)
  EXIT    time    0.005s  (  0.004s elapsed)
  Total   time  628.749s  ( 70.726s elapsed)

  Alloc rate    1,158,708,427 bytes per MUT second

  Productivity  87.7% of total user, 80.8% of total elapsed

./parallel_icm +RTS -N12 -ls -s  609.01s user 19.75s system 888% cpu 1:10.75 total
```

This run utilized 12 capabilities, creating 12 sparks to parallelize the computations. Of these, 11 sparks converted into useful parallel tasks, while one fizzled (became unnecessary). Our program still effectively parallelized the workload of multiple Monte Carlo simulations.

6.3 Optimized Parallel ICM Threadscope



From Threadscope, it becomes apparent that we are getting very consistent CPU usage, indicating that the workload is well balanced amongst each of the available cores. This is a further sign of the successful parallelization of the ICM model's Monte Carlo simulations.

Appendix A: Graph Construction Details

Here we provide the details of the graph construction process used in our experiments. The pseudocode and functions below complement the main text, offering a thorough explanation of how the nodes and edges are generated.

Node and Edge Setup

We begin with N nodes, labeled from 0 to $N-1$. The graph is constructed by generating E unique directed edges. Each edge is assigned a probability weight chosen uniformly at random within the range $[0.1, 0.5]$.

Algorithm 3 Graph Construction Pseudocode

```
1:  function generateUniqueEdges(n):
2:      S ← ∅
3:      while |S| < n do:
4:          i ← randomInt(0, N − 1)
5:          j ← randomInt(0, N − 1)
6:          if i ≠ j then:
7:              S ← S ∪ {(i, j)}
8:          end if
9:      end while
10:     return list(S)
11: end function
12: function generateWeightedEdges(n):
13:     edges ← generateUniqueEdges(n)
14:     weightedEdges ← []
15:     for each (i, j) ∈ edges do:
16:         w ← randomDouble(0.1, 0.5)
17:         append (i, j, w)  to  weightedEdges
18:     end for
19:     return weightedEdges
20: end function
21: function buildGraph:
22:     nodes ← {(0, ()), (1, ()), . . . , (N − 1, ())}
23:     edges ← generateWeightedEdges(E)
24:     graph ← mkGraph(nodes, edges)
25:     return graph
26: end function
```

This pseudocode ensures that the graph contains no self-loops and that all edges are unique. The function `buildGraph` outputs a weighted directed graph suitable for subsequent Monte Carlo simulations and Independent Cascade Model analyses discussed in the main text.

Appendix B: Naïve Parallel ICM

```
~/De/Col/1/shocknet main !2 ?13 ❯ ./unoptimized_parallel_icm +RTS -N12 -s
M
Building graph...
Graph built. Running Monte Carlo simulations...
Average Influence (Monte Carlo): 26563.16
 573,383,456,056 bytes allocated in the heap
  34,712,871,544 bytes copied during GC
      95,884,456 bytes maximum residency (154 sample(s))
       1,819,128 bytes maximum slop
             327 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     23586 colls, 23586 par   27.891s   9.761s     0.0004s    0.0166s
  Gen  1       154 colls,   153 par   12.100s   1.506s     0.0098s    0.0194s

  Parallel GC work balance: 66.07% (serial 0%, perfect 100%)

  TASKS: 26 (1 bound, 25 peak workers (25 total), using -N12)

  SPARKS: 12 (11 converted, 0 overflowed, 0 dud, 0 GC'd, 1 fizzled)

  INIT    time    0.008s  (  0.010s elapsed)
  MUT     time  2826.848s  (792.032s elapsed)
  GC      time   39.991s  ( 11.268s elapsed)
  EXIT    time    0.005s  (  0.009s elapsed)
  Total   time  2866.852s  (803.318s elapsed)

  Alloc rate    202,834,928 bytes per MUT second

  Productivity  98.6% of total user, 98.6% of total elapsed
```

This is the output of running the naïve parallel ICM where we did not parallelize random number generation.

Appendix C: Project Code

Sequential ICM

```
import Data.Graph.Inductive.PatriciaTree (Gr)
import Data.Graph.Inductive.Graph
import Data.Maybe (catMaybes)
import qualified Data.Set as Set
import System.Random
import Control.Monad
import Data.List (foldl')

type SimpleEdge = (Int, Int)


-- TESTING PARAMETERS --

nodesCount :: Int
nodesCount = 30000

edgesCount :: Int
edgesCount = 300000

numSimulations :: Int
numSimulations = 1000


-- MAIN METHOD --

{-|
  Main method:
    - builds the graph with specified nodesCount & edgesCount
    - executes numSimulations number of monte carlo simulations of
       the ICM using the seed
       set (in this case node 0)
    - displays resulting average influence
-}
main :: IO ()
main = do
    putStrLn "Building graph..."
    graph <- buildGraph
    putStrLn "Running Monte Carlo simulations..."
    let seedNodes = [0]
    averageInfluence <- monteCarloSimulation graph seedNodes
       numSimulations
    putStrLn $ "Average Influence (Monte Carlo): " ++ show
       averageInfluence
```

```
41
42
43  -- METHODS FOR BUILDING THE GRAPH --
44
45  {-|
46      Builds a list of unique edges (without repetition or
          self-looping)
47      Basically, we pick any random two nodes, and if no edge exists
          between them,
48      we add the new edge to a set.
49  -}
50
51  generateUniqueEdges :: Int -> IO [SimpleEdge]
52  generateUniqueEdges n = do
53      let loop s
54              | Set.size s >= n = return (Set.toList s)
55              | otherwise = do
56                  i <- randomRIO (0, nodesCount-1)
57                  j <- randomRIO (0, nodesCount-1)
58                  if i /= j then
59                      let s' = Set.insert (i,j) s
60                      in loop s'
61                  else loop s
62      loop Set.empty
63
64
65  {-|
66      This method assigns weights to each edge. We pick the weights
          for each edge
67      picking a random float between [0.1, 0.5]
68  -}
69
70  generateWeightedEdges :: Int -> IO [LEdge Double]
71  generateWeightedEdges n = do
72      edges <- generateUniqueEdges n
73      forM edges $ \(i, j) -> do
74          weight <- randomRIO (0.1, 0.5)
75          return (i, j, weight)
76
77  {-|
78    Build a directed graph with a specified number of nodes and a
        specified number
79    of randomly generated edges.
80  -}
81
82  buildGraph :: IO (Gr () Double)
83  buildGraph = do
```

```
84     let nodes = [(i, ()) | i <- [0..nodesCount-1]]
85     edges <- generateWeightedEdges edgesCount
86     return $ mkGraph nodes edges
87
88
89 -- INDEPENDENT CASCADE AND MONTE CARLO SIMULATIONS
90
91 {-|
92   Perform one run of ICM given a graph and a set of initially
       activated nodes (seeds).
93   Refer to the Problem formulation in our report for an explanation
       of the ICM model
94 -}
95 -- note here we aren't using stdGen
96 independentCascade :: Gr () Double -> [Node] -> IO (Set.Set Node)
97 independentCascade graph seeds = go (Set.fromList seeds)
     (Set.fromList seeds)
98   where
99     go :: Set.Set Node -> Set.Set Node -> IO (Set.Set Node)
100     go activatedNodes newlyActivated
101         | Set.null newlyActivated = return activatedNodes
102         | otherwise = do
103             nextActivatedList <- forM (Set.toList newlyActivated) $
                   \node -> do
104                 let neighbors = lsuc graph node
105                 activatedNeighbors <- forM neighbors $ \(neighbor,
                       weight) -> do
106                     if neighbor `Set.member` activatedNodes
107                         then return Nothing
108                         else do
109                             r <- randomRIO (0.0, 1.0 :: Double)
110                             if r <= weight
111                                 then return $ Just neighbor
112                                 else return Nothing
113                 return $ catMaybes activatedNeighbors
114             let nextActivated = Set.fromList $ concat
                   nextActivatedList
115             let activatedNodes' = Set.union activatedNodes
                   nextActivated
116             go activatedNodes' nextActivated
117
118
119 {-|
120   Perform a Monte Carlo simulations
121
122   The simulation repeats the Independent Cascade process a
       specified number of times
```

```
123   (`numSimulations`). This function returns the average influence
         over each simulation.
124
125   This version uses replicateM to run the simulation multiple times
126   sequentially and accumulate the results.
127 -}
128 monteCarloSimulation :: Gr () Double -> [Node] -> Int -> IO Double
129 monteCarloSimulation graph seeds numSimulations = do
130     totalActivated <- replicateM numSimulations $ do
131         activatedNodes <- independentCascade graph seeds
132         return $ fromIntegral $ Set.size activatedNodes
133     let total = sum totalActivated
134     return $ total / fromIntegral numSimulations
```

Naïve Parallel ICM

```
import Data.Graph.Inductive.PatriciaTree (Gr)
import Data.Graph.Inductive.Graph
import Data.Maybe (catMaybes)
import qualified Data.Set as Set
import System.Random
import Control.Monad
import Data.List (foldl')
import Control.Parallel.Strategies
import GHC.Conc (getNumCapabilities)
import System.IO.Unsafe (unsafePerformIO)

type SimpleEdge = (Int, Int)

-- TESTING PARAMETERS --
nodesCount :: Int
nodesCount = 30000

edgesCount :: Int
edgesCount = 300000

numSimulations :: Int
numSimulations = 1000

{-|
  Main method:
    - builds the graph with specified nodesCount & edgesCount
    - executes numSimulations number of monte carlo simulations of
       the ICM using the seed
      set (in this case node 0)
    - displays resulting average influence
-}
main :: IO ()
main = do
    putStrLn "Building graph..."
    graph <- buildGraph
    putStrLn "Graph built. Running Monte Carlo simulations..."
    let seedNodes = [0]
    averageInfluence <- monteCarloSimulation graph seedNodes
       numSimulations
    putStrLn $ "Average Influence (Monte Carlo): " ++ show
       averageInfluence

-- METHODS FOR BUILDING THE GRAPH --

generateUniqueEdges :: Int -> IO [SimpleEdge]
generateUniqueEdges n = do
```

```
44      let loop s
45            | Set.size s >= n = return (Set.toList s)
46            | otherwise = do
47                i <- randomRIO (0, nodesCount-1)
48                j <- randomRIO (0, nodesCount-1)
49                if i /= j then
50                    let s' = Set.insert (i,j) s
51                    in loop s'
52                else loop s
53      loop Set.empty
54
55
56  generateWeightedEdges :: Int -> IO [LEdge Double]
57  generateWeightedEdges n = do
58      edges <- generateUniqueEdges n
59      forM edges $ \(i, j) -> do
60          weight <- randomRIO (0.1, 0.5)
61          return (i, j, weight)
62
63
64  buildGraph :: IO (Gr () Double)
65  buildGraph = do
66      let nodes = [(i, ()) | i <- [0..nodesCount-1]]
67      edges <- generateWeightedEdges edgesCount
68      return $ mkGraph nodes edges
69
70
71  {-| Actually executes the independent cascade model very
       inefficiently by using IO-based randomization on every step. -}
72  independentCascade :: Gr () Double -> [Node] -> IO (Set.Set Node)
73  independentCascade graph seeds = go (Set.fromList seeds)
       (Set.fromList seeds)
74    where
75      go :: Set.Set Node -> Set.Set Node -> IO (Set.Set Node)
76      go activatedNodes newlyActivated
77          | Set.null newlyActivated = return activatedNodes
78          | otherwise = do
79              nextActivatedList <- forM (Set.toList newlyActivated) $
                   \node -> do
80                 let neighbors = lsuc graph node
81                 activatedNeighbors <- forM neighbors $ \(neighbor,
                       weight) -> do
82                     if neighbor `Set.member` activatedNodes
83                         then return Nothing
84                         else do
85                             r <- randomRIO (0.0, 1.0 :: Double)
86                             if r <= weight
```

```
87                                 then return $ Just neighbor
88                                 else return Nothing
89                     return $ catMaybes activatedNeighbors
90                 let nextActivated = Set.fromList $ concat
                    nextActivatedList
91                 let activatedNodes' = Set.union activatedNodes
                    nextActivated
92                 go activatedNodes' nextActivated
93 
94 simulateOnce :: Gr () Double -> [Node] -> Double
95 simulateOnce g s = unsafePerformIO $ do
96     activatedNodes <- independentCascade g s
97     return $ fromIntegral $ Set.size activatedNodes
98 
99 monteCarloSimulation :: Gr () Double -> [Node] -> Int -> IO Double
100 monteCarloSimulation graph seeds numSims = do
101     numCapabilities <- getNumCapabilities
102     let chunkSize = (numSims + numCapabilities - 1) `div`
        numCapabilities
103     let workChunks = replicate numCapabilities (replicate chunkSize
        ())
104 
105     let results = parMap rdeepseq (\chunk ->
106                     sum [ simulateOnce graph seeds | _ <- chunk ]
107                   ) workChunks
108 
109     let totalActivated = sum results
110     let averageInfluence = totalActivated / fromIntegral numSims
111     return averageInfluence
112 
113 chunkList :: Int -> [a] -> [[a]]
114 chunkList _ [] = []
115 chunkList n xs = take n xs : chunkList n (drop n xs)
```

Parallel ICM

```
import Data.Graph.Inductive.PatriciaTree (Gr)
import Data.Graph.Inductive.Graph
import Data.Maybe (catMaybes)
import qualified Data.Set as Set
import System.Random
import Control.Monad
import Control.Monad.Random
import Data.List (foldl')
import Control.Parallel.Strategies
import GHC.Conc (getNumCapabilities)

type SimpleEdge = (Int, Int)


-- TESTING PARAMETERS --

nodesCount :: Int
nodesCount = 30000

edgesCount :: Int
edgesCount = 300000

numSimulations :: Int
numSimulations = 1000

-- MAIN METHOD --

{-|
  Main method:
    - builds the graph with specified nodesCount & edgesCount
    - executes numSimulations number of monte carlo simulations of
       the ICM using the seed
       set (in this case node 0)
    - displays resulting average influence
-}
main :: IO ()
main = do
    putStrLn "Building graph..."
    graph <- buildGraph
    putStrLn "Graph built. Running Monte Carlo simulations..."
    let seedNodes = [0]
    averageInfluence <- monteCarloSimulation graph seedNodes
       numSimulations
    putStrLn $ "Average Influence (Monte Carlo): " ++ show
       averageInfluence

```

```
-- METHODS FOR BUILDING THE GRAPH --

{-|
    Builds a list of unique edges (without repittion or
       self-looping)
    Basically, we pick any random two nodes, and if no edge exists
       between them,
    we add the new edge to a set.
-}

generateUniqueEdges :: Int -> IO [SimpleEdge]
generateUniqueEdges n = do
    let loop s
          | Set.size s >= n = return (Set.toList s)
          | otherwise = do
              i <- randomRIO (0, nodesCount-1)
              j <- randomRIO (0, nodesCount-1)
              if i /= j then
                  let s' = Set.insert (i,j) s
                  in loop s'
              else loop s
    loop Set.empty


{-|
    This method assigns weights to each edge. We pick the weights
       for each edge
    picking a random float between [0.1, 0.5]
-}

generateWeightedEdges :: Int -> IO [LEdge Double]
generateWeightedEdges n = do
    edges <- generateUniqueEdges n
    forM edges $ \(i, j) -> do
        weight <- randomRIO (0.1, 0.5)
        return (i, j, weight)

{-|
  Build a directed graph with a specified number of nodes and a
     specified number
  of randomly generated edges.
-}

buildGraph :: IO (Gr () Double)
buildGraph = do
    let nodes = [(i, ()) | i <- [0..nodesCount-1]]
```

```haskell
    edges <- generateWeightedEdges edgesCount
    return $ mkGraph nodes edges


-- INDEPENDENT CASCADE AND MONTE CARLO SIMULATIONS

{-|
  Perform one run of ICM given a graph and a set of initially activated nodes (seeds).
  Refer to the Problem formulation in our report for an explanation of the ICM model
-}
-- note here we are using stdGen
independentCascade :: Gr () Double -> [Node] -> Rand StdGen (Set.Set Node)
independentCascade graph seeds = go (Set.fromList seeds) (Set.fromList seeds)
  where
    go :: Set.Set Node -> Set.Set Node -> Rand StdGen (Set.Set Node)
    go activatedNodes newlyActivated
        | Set.null newlyActivated = return activatedNodes
        | otherwise = do
            nextActivatedList <- forM (Set.toList newlyActivated) $ \node -> do
                let neighbors = lsuc graph node
                activatedNeighbors <- forM neighbors $ \(neighbor, weight) -> do
                    if neighbor `Set.member` activatedNodes
                        then return Nothing
                        else do
                            r <- getRandomR (0.0, 1.0 :: Double)
                            if r <= weight
                                then return $ Just neighbor
                                else return Nothing
                return $ catMaybes activatedNeighbors
            let nextActivated = Set.fromList $ concat nextActivatedList
            let activatedNodes' = Set.union activatedNodes nextActivated
            go activatedNodes' nextActivated


{-|
  Perform a Monte Carlo simulation of the Independent Cascade model:
    - given a graph and seed nodes, run the ICM multiple times
    - calculate the average influence of the seed set over the number of simulations.
```

```haskell

  Parallelization:
  The simulation uses parallel strategies to partition the runs
      among available
  CPU cores. It splits random number generators per core and
      utilizes static chunking.

  returns the average number of "infected" nodes
-}

monteCarloSimulation :: Gr () Double -> [Node] -> Int -> IO Double
monteCarloSimulation graph seeds numSims = do
    numCapabilities <- getNumCapabilities
    stdGen <- getStdGen
    let gens = take numSims $ iterate (snd . split) stdGen
    let chunkSize = (numSims + numCapabilities - 1) `div`
       numCapabilities
    let chunks = chunkList chunkSize gens
    let results = parMap rdeepseq (\genChunk ->
                        sum [ evalRand (simulateOnce graph seeds) gen |
                           gen <- genChunk ]
                       ) chunks

    let totalActivated = sum results
    let averageInfluence = totalActivated / fromIntegral numSims
    return averageInfluence

  where
    -- actually executes the independent cascade model once
    simulateOnce :: Gr () Double -> [Node] -> Rand StdGen Double
    simulateOnce g s = do
        activatedNodes <- independentCascade g s
        return $ fromIntegral $ Set.size activatedNodes

    -- splits workload into chunks
    chunkList :: Int -> [a] -> [[a]]
    chunkList _ [] = []
    chunkList n xs = take n xs : chunkList n (drop n xs)
```

References

[1] Pedro Domingos and Matt Richardson. “Mining the network value of customers”. In: *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2001, pp. 57–66.