Project Report: Hascade

Yiming Fang
Email: yf2484@columbia.edu

December 22, 2022

1 Introduction

Influence Maximization (IM) is an exciting and well-researched topic, and it has practical applications to commercial marketing and social network management. In any given social network, it often is the case that some nodes are more influential than others. Sometimes, identifying a set of influential nodes can greatly help decision-makers make marketing decisions. However, since the problem is proven to be NP-Hard, most existing algorithms use greedy heuristics that run sequentially. Although the approximation algorithms have theoretical error bounds, the computational cost of IM solvers is usually very high due to their sequential nature.

This project presents a parallel implementation of the IM solver. In particular, we focus on the IM problem in the context of the Independent Cascade model. The following sections are going to be divided as follows. Section 2 formulates the problem. Section 3 describes the greedy algorithm we used. Section 4 discusses the choices we made for parallelization. Section 5 presents the empirical results of the performance.

2 Problem Formulation

Given a graph $G = (V, E)$, the task of the IM problem identifying a "most influential seed set" $S \subseteq V$ of size k, such that

$$S = \arg\max_{|S|=k} E[f(S)]$$

where S is called the seeds, and $f(S)$ is the total expected cascade size resulting from S.

In the Independent Cascade model, each edge $e_{(u,v)}$ has an influence probability $p_{u,v}$, such that u has a one-shot opportunity of influencing v with $p_{u,v}$ probability.

The influence process runs as follows. In the beginning, only the seeds are activated. At each timestep, the activated vertices have the opportunity to influence their neighbors, and if they succeed in influencing a vertex, the new vertex will join the set of activated vertices in the next iteration. The process terminates when there all vertices are either activated or have been tried to be activated by some neighbor.

3 Implementation

The baseline implementation of IM is a sequential, greedy algorithm [1] [2]. The algorithm adds vertices to the "most influential set" one-by-one, by always choosing the vertex that brings the highest expected increase of influence to the current set. The algorithm terminates when all k slots are populated. We present this process in Algorithm 1.

Algorithm 1 Greedy

1: **Input**: Graph $G = (V, E)$.
2: **Output**: Most Influential Set S.
3:
4: $S_0 \leftarrow \{\}$
5: **for** $i = 1, ..., k$, **do**
6: $\quad u \leftarrow \arg\max_u f(S_{i-1} \cup \{u\})$
7: $\quad$ Activate u
8: $\quad S_i \leftarrow S_{i-1} \cup \{u\}$
9: **end**
10:
11: **return** S_k
12: =0

The most important and time-consuming part of this algorithm is line 6, which computes an approximation of the expected influence of a set of activated vertices. This approximation can be done through a simple Monte Carlo-style search that runs sequentially.

Algorithm 2 Monte-Carlo

1: **Input**: Graph $G = (V, E)$ Vertex set $S_i = S_{i-1} \cup \{u\}$, number of trails N.
2: **Output**: The expected influence of S_i
3:
4: count $\leftarrow 0$
5: **for** $j = 1, ..., N$, **do**
6: $\quad$ Simulate independent cascade on G, S_i
7: $\quad$ count += number of influenced vertex
8: **end**
9:
10: **return** count / N.
11: =0

One obvious thing to notice is that we are running the independent function calls over the exact same input vertex set for a large number of iterations, and this can be easily made parallel, as discussed in the next section.

The simulation of the independent cascade model is done by finding the all neighbors of every vertex in the input set, and trying to activate them by generating a random number and comparing it with the influence probability $p_{u,v}$. Activated vertices are added to the input set of the recursive call to the next simulation, and vertices that have been attempted but not successfully activated are removed from the graph given to the recursive call.

Algorithm 3 Independent-Cascade

Input: Graph $G = (V, E)$ Vertex set S_i.
Output: The simulated influence of S_i

neighbors $\leftarrow \cup_{v \in S}\ G.lookup(v)$
neighbors = neighbors \ S
activated $\leftarrow$ tryActivate(neighbors)
activated = activated \ S
failed = neighbors \ activated
$G.keys \leftarrow G.keys \backslash$ failed

return length(activated) + Independent-Cascade($G, S \cup$ activated)
=0

4 Parallelization

As we observed in the last section, the most appropriate place to introduce parallelism is in Algorithm 3, because the structure of the sequential algorithm can be modified very slightly to bring parallelism to the overall algorithm. Since we used lazy data structures in our code, we want to ensure that our expressions are all fully evaluated to the normal form. We can accomplish this by using the "rdeepseq" strategy. The modified parallel Monte Carlo algorithm is presented as follows.

Algorithm 4 Monte-Carlo Par

Input: Graph $G = (V, E)$ Vertex set $S_i = S_{i-1} \cup \{u\}$, number of trails N.
Output: The expected influence of S_i

count $\leftarrow 0$
chunks $\leftarrow$ split numCores N
for *chunk* $\in$ *chunks*, **do**
    for $j = 1, ..., N/numCores$, **do**
        Simulate independent cascade on G, S_i
        count += number of influenced vertex
    end
end 'using' parList rdeepseq
return count / N.
=0

A further parallelization trick that we used to make the algorithm more efficient is static chunking. While it is in general more beneficial to use dynamic partitioning to initialize sparks, we observe that in the situation of Monte Carlo, static partitioning suffices, and even outperforms dynamic partitioning.

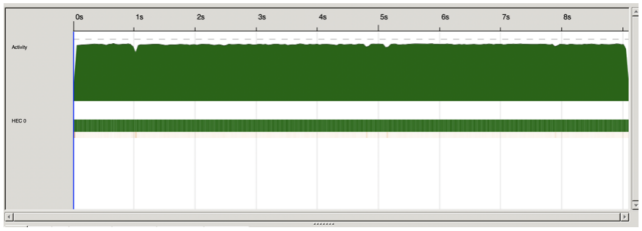
For each function call to Monte-Carlo, we are going to split the number of trials into numCores chunks, and start a spark for each chunk.

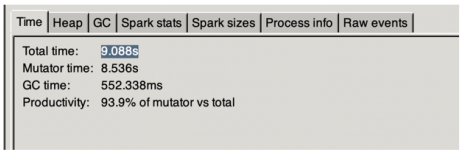
The rationale for doing so is the following. In most cases, whenever we do Monte Carlo simulation, we would like to ensure that the approximation we get is both stable and accurate. To this end,

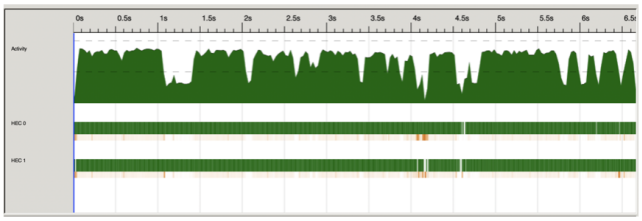
the number of trials for Monte Carlo is going to be always a large number. Therefore, when we divide the workload into chunks, although each Independent-Cascade simulation can differ a lot in terms of workload and compute time, the chunk of simulation should all have similar workloads. This property ensures that the sparks started for the same Monte Carlo function call should finish roughly at the same time, minimizing the idle time of cores waiting for other cores to finish.

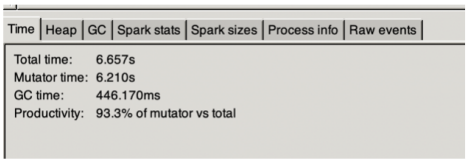
5 Performance

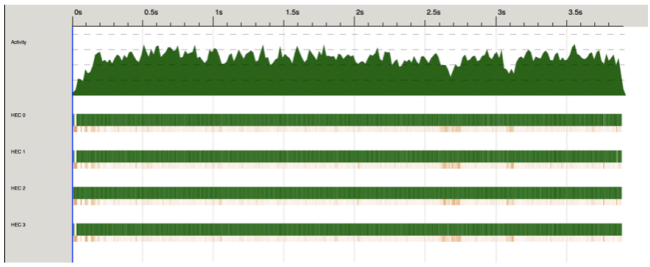
For testing, we used a dataset consisting of 1000 vertices, and set the hyper-parameters of Monte Carlo trials to be 1000. I used a 2018 MacBook Pro with 4 Intel cores and 8GB RAM, and I tested for sequential, 2 cores, and 4 cores respectively. The results is summarized in the following screenshot from ThreadScope.

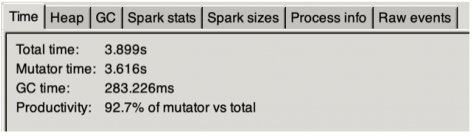






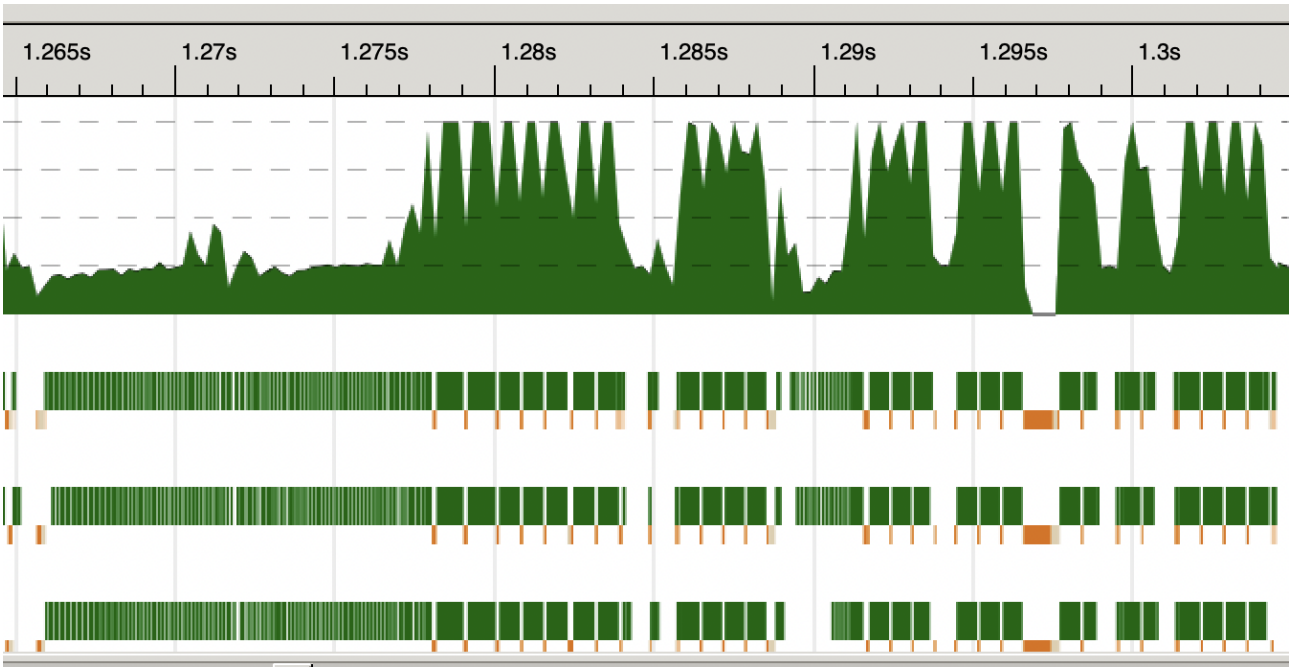






As one can see from the picture, the multi-core tests achieved reasonable speedup ratios over the sequential version of the algorithm. For 2 cores, the speedup is **1.47x**, and for 4 cores the speedup is **2.33x**.

From the trace of the sparks, one can also see that the tasks are well-balanced, although there are periods of time where the balance is not so great and the performance becomes near sequential. To see exactly what happened during these sections, we can zoom into look at a more microscopic image:



In this test run with 4 cores, one can observe that the beginning section is most likely doing some sequential operations, such as chunking, combining, and folding. During this time, the cores are not fully occupied, and the stack trace tells us that the threads are blocked by some other threads, waiting for some relevant execution to finish. At arround 1.275 seconds, the cores begin to do more meaningful work, interrupted periodically by garbage collections.

Given the nature of this algorithm, the degree of parallelization can be affected by a lot of factors, and can also differ across input. For example, a dense graph would cause the algorithm to spend more time doing Independent-Cascade simulation, resulting in a better ratio of parallelization. Moreover, the influence probability is also positively related to the ratio of speedup. To make a fair comparison, we have used a real-world graph from the SNAP datasets, representing the wikipedia community [3]

6 Conde Listing

```haskell
{-# OPTIONS_GHC -Wno-missing-export-lists #-}
module Main where

import           BasicTypes                      ( UnweightedGraph )
import qualified Data.Map.Strict                 as Map
import qualified Data.Set                        as Set
import           Solver                          ( greedySolver )
import           System.Environment              ( getArgs
                                                 , getProgName
                                                 )
import           System.Exit                     ( die )


main :: IO ()
main = do
```

```haskell
  args <- getArgs
  case args of
    [k, filename] -> do
      contents <- readFile filename
      let inputGraph = constructGraph contents
      print $ greedySolver inputGraph Set.empty (read k) 0.1 100
    _ -> do
      pn <- getProgName
      die $ "Usage: " ++ pn ++ "<num_cores> <filename>"


constructGraph :: String -> UnweightedGraph
constructGraph = Map.fromList . map extractLine . lines


extractLine :: String -> (Int, [Int])
extractLine str = (node, neighbors)
 where
  (node, neighbors) = case words str of
    (this : others) -> (read this, map read others)
    []              -> (-1, [])
```

Listing 1: Main.hs

```haskell
{-# OPTIONS_GHC -Wno-unrecognised-pragmas #-}
{-# HLINT ignore "Use newtype instead of data" #-}
module BasicTypes
    ( Vertex
    , Weight
    , UnweightedGraph
    , WeightedGraph
    ) where


import qualified Data.Map.Strict               as Map

type Vertex = Int
type Weight = Float

type UnweightedGraph = Map.Map Vertex [Vertex]
type WeightedGraph = Map.Map Vertex [(Vertex, Weight)]
```

Listing 2: BasicTypes.hs

```haskell
{-# LANGUAGE BlockArguments #-}
module Solver
      ( greedySolver
      ) where

import           BasicTypes                    ( UnweightedGraph
                                               , Vertex
                                               )
import           Control.Monad                 ( replicateM )
import           Control.Parallel.Strategies   ( parList
                                               , rdeepseq
                                               -- , rpar
                                               -- , rseq
                                               , using
                                               )
```

```
import qualified Data.Map.Strict                as Map
import           Data.Maybe                     ( fromMaybe )
import qualified Data.Set                       as Set
import           Data.Set                       ( Set
                                                , (\\)
                                                )
import           System.IO.Unsafe               ( unsafePerformIO )
import           System.Random                  ( randomIO )



greedySolver
      :: UnweightedGraph -> Set Vertex -> Int -> Float -> Int -> Set Vertex
greedySolver graph vSet k thresh mcTrials
      | k == 0
      = vSet
      | otherwise
      = let runMC :: Vertex -> (Float, Vertex)
            runMC = monteCarlo graph vSet buffer

            runChunk :: [Vertex] -> [(Float, Vertex)]
            runChunk vs = map runMC vs

            findMaxV :: [(Float, Vertex)] -> (Float, Vertex) -> Vertex
            findMaxV [] acc = snd acc
            findMaxV (x : xs) acc | fst x > fst acc = findMaxV xs x
                                  | otherwise       = findMaxV xs acc

            buffer      = replicate mcTrials thresh
            candidateVs = Map.keys graph

            -- candidateChunks = split 10 candidateVs
            -- scores      = map runChunk candidateChunks `using` parList rdeepseq
            -- vMax        = findMaxV (concat scores) (0, -1)

            scores      = map runMC candidateVs -- `using` parList rdeepseq
            vMax        = findMaxV scores (0, -1)

            vSet'       = Set.insert vMax vSet
        in  greedySolver graph vSet' (k - 1) thresh mcTrials


split :: Int -> [a] -> [[a]]
split numChunks xs = chunk (length xs `quot` numChunks) xs


chunk :: Int -> [a] -> [[a]]
chunk _ [] = []
chunk n xs = let (as, bs) = splitAt n xs in as : chunk n bs


monteCarloV1
      :: UnweightedGraph -> Set Vertex -> [Float] -> Int -> (Float, Vertex)
monteCarloV1 graph vSet ps vNew = (mean, vNew)
   where
      vs   = Set.insert vNew vSet
      lens = map (independentCascade graph vs 0) ps `using` parList rdeepseq
      mean = sum lens / realToFrac (length lens)

```

```haskell

monteCarlo
      :: UnweightedGraph -> Set Vertex -> [Float] -> Vertex -> (Float, Vertex)
monteCarlo graph vSet ps vNew = (mean, vNew)
   where
      meansWithSizes = map mc pss `using` parList rdeepseq

      vs             = Set.insert vNew vSet
      pss            = split 4 ps

      mc             = monteCarloChunk graph vs

      totalSum       = foldr ((+) . multSize) 0 meansWithSizes
      totalSize      = foldr ((+) . snd) 0 meansWithSizes

      multSize (a, b) = a * realToFrac b
      mean = totalSum / realToFrac totalSize


monteCarloChunk :: UnweightedGraph -> Set Vertex -> [Float] -> (Float, Vertex)
monteCarloChunk graph vSet ps = (mean, length ps)
   where
      lens = map (independentCascade graph vSet 0) ps -- `using` parList rseq
      mean = sum lens / realToFrac (length lens)


independentCascade :: UnweightedGraph -> Set Vertex -> Int -> Float -> Float
independentCascade graph vSet depth thresh = if null activatedSet'
      then 0
      else
            let nextCascade =
                      independentCascade graph' activeSet (depth + 1) thresh
                thisCascade = realToFrac $ length activeSet
            in  thisCascade + nextCascade

   where
      graph'        = graph Map.\\ setMap
      setMap        = Map.fromSet (`Map.lookup` graph) neighborSet'

      neighborSet   = getNeiborSet graph vSet
      neighborSet'  = neighborSet \\ vSet

      activatedSet  = tryActivate neighborSet' thresh
      activatedSet' = activatedSet \\ vSet

      activeSet     = Set.union vSet activatedSet'


randSeq :: Int -> [Float]
randSeq k = unsafePerformIO (replicateM k (randomIO :: IO Float))


tryActivate :: Set Vertex -> Float -> Set Vertex
tryActivate vs thresh = Set.fromList newActiveVs
   where
      strengths   = randSeq l
      threshs     = replicate l thresh
      l           = length vs

```

```
      diff         = zipWith (-) threshs strengths
      threshVs     = zip diff (Set.toList vs)
      newActive    = filter ((< 0) . fst) threshVs

      newActiveVs = map snd newActive


getNeiborSet :: UnweightedGraph -> Set Vertex -> Set Vertex
getNeiborSet graph vSet = Set.fromList newSets
   where
      findChildren :: Int -> [Int]
      findChildren v = Data.Maybe.fromMaybe [] (Map.lookup v graph)

      newSets = concatMap findChildren $ Set.toList vSet
```

Listing 3: Solver.hs

References

[1] https://snap-stanford.github.io/cs224w-notes/network-methods/influence-maximization

[2] https://hautahi.com/im_greedycelf

[3] https://snap.stanford.edu/data/wiki-Talk.html