1 Introduction

Currently, there is a boosting trend of users size in social applications because of the popularization of personal smart devise. This constructs huge social networks containing considerably large quantities of interactions among those users, which is worth further investigation for more accurate and intelligent future functionality. One task is to measure how significant, within a graph, a vertex is. Many algorithms are reported including Betweenness Centrality which our project focuses on.

However, in reality, the network is highly likely vast in size. This could be changing to apply Betweenness Centrality algorithm on those real-world data. Therefore, the project aims to provide a parallel implementation of Betweenness Centrality Algorithm to make the it efficient to perform centrality analysis in real-world social network.

2 Problem Definition

Betweenness Centrality is formulated by Freeman [1] as shown below.

\[ Betweenness(k) = \sum_{i \neq k \neq j} \left( \frac{\sigma_{i,j}(k)}{\sigma_{i,j}} \right) \]

Where \( \sigma_{i,j}(k) \) is the number of shortest path between \( i, j \) containing the node \( k \) and \( \sigma_{i,j} \) is the total number of shortest path between \( i, j \). This could, to some extend, reflect the centrality of a node within a network. That is because a node on the shortest path from \( i \) to \( j \) is usually more central than one that is not.
could be regarded as a bridge connecting those two vertices and its significance should be proportional to the total number of shortest path passing through it. For example, with regards to social network, a node at the center of the whole network or some major components or a node acting like a gateway between two major groups will have higher betweenness centrality.

3 Implementation and Algorithm Design

3.1 Notation

3.1.1 Graph Presentation

To take advantage of the sparse nature of social networks, instead of using adjacent matrix, adjacent lists are used. It is implemented by Map in Haskell, presenting like:

\{source : \{neighbour_1, neighbour_2, ...\}\}

where the key source is an integer identifying a node and the value is a list of integers identifying all its neighbours.

3.1.2 Algorithm Design

We started with the original algorithm we proposed at the beginning. The algorithm is divided into two steps. First step is calculating pair-wise shortest distances and the second is quantifying the times of occurrence of a node on those shortest paths. The heavy calculation inside each function can be implemented in a parallel fashion while two steps should be strictly sequential to ensure correctness of algorithm.

We are targeting at analyzing the betweenness centrality of github network which is inherently a unweighted and undirected graph with our ParBC algorithm in the end. We get insights from Breath-First Search for calculating the pair-wise shortest path. Every BFS-like shortest path calculation can be applied to a single node which provides great feasibility of parallelizing. To acquire the betweenness for each node, the most heavily computational part is calculating the number of shortest path passing through a particular node. It’s really expensive if we trying to record every path and extract betweenness from that. Instead, we gain insights from Floyd–Warshall algorithm \cite{2}. More explicitly, for a node v, if the summation of shortest path length from i to v and shortest path length from v to j is equal to the shortest path length from i to j, then we can conclude the shortest path from i to j pass through v. However,
there might be two major limitations. First, not all things can be fully parallelized. The execution of betweenness calculation is depend on the result from shortest path length calculation. This forces the threads to be synchronised and wait in this loop. Second, its complexity is $O(n^3)$ where $n$ is the number of nodes. However, in reality, social network is edge-sparse but the algorithm could not take a good advantage of this feature. Therefore, we turn to Brandes’ algorithm [3], which utilizes sparsity of social network quite well. Comparing to $O(n^3)$ for original proposal, it has $O(nm)$, where $m$ is the number of edges in the graph (where $n << m$).

To quantify the times of occurrence of a node on those shortest paths, we could use map-and-reduce fashion to perform this calculation in a parallel manner where, node id is the key and count is the value. This could make a good use of parallelism to calculate the final betweenness centrality for all the nodes in a network.

### 3.1.3 Brandes’ Algorithm

To describe Brandes’ Algorithm [3] and our further adjustment, its notation and precisely formulation would be shown in the following two paragraphs.

First, we follow the notation from the definition of Betweenness Centrality where $\sigma_{i,j}(k)$ is the number of shortest path between $i, j$ containing the node $k$ and $\sigma_{i,j}$ is the total number of shortest path between $i, j$. In addition, predecessor nodes is defined as:

\[
pred_s(v) = \{w \in V | \{w, v\} \in E, d(s, v) = d(s, w) + \omega(w, v)\}
\]

Given that the graph is unweighted, $\omega(w, v) = 1$. Therefore, as mentioned in Section 3.1.2, BFS could be utilized to calculate $\sigma_{i,j}$ with $\rho_{d_s(v)}$ as shown below:

\[
\sigma_{i,j} = \sum_{w \in \text{Pred}_s(j)} \sigma_{i,w}
\]

In addition, $\rho_{d_s(v)}$ helps in calculating betweenness. To simplify the formulation, it denotes the number of shortest path from $s$ to $t$ passing through node $v$ and edge $v, w$ is $\sigma_{s,t}(v, \{v, w\})$ and:

\[
\delta_{s,t}(v, \{v, w\}) = \frac{\sigma_{s,t}(v, \{v, w\})}{\sigma_{s,t}}
\]

\[
\delta_s(v) = \sum_{t \in V} \sum_{w \in \text{Pred}_s(v)} \delta_{s,t}(v, \{v, w\}) = \sum_{w \in \text{Pred}_s(v)} \sum_{t \in V} \delta_{s,t}(v, \{v, w\})
\]
so if \( w \neq t \)

\[
\delta_{s,t}(v, \{v, w\}) = \frac{\sigma_{s,v} \sigma_{s,t}(w)}{\sigma_{s,w}}
\]

if \( w = t \)

\[
\delta_{s,t}(v, \{v, w\}) = \frac{\sigma_{s,v}}{\sigma_{s,w}}
\]

Therefore

\[
\delta_{s}(v) = \sum_{w \in \text{Pred}_s(v)} \sum_{t \in V} \delta_{s,t}(v, \{v, w\}) = \sum_{w \in \text{Pred}_s(v)} \left( \frac{\sigma_{s,v}}{\sigma_{s,w}} + \sum_{t \in V, t \neq w} \frac{\sigma_{s,v} \sigma_{s,t}(w)}{\sigma_{s,w}} \right) = \sum_{w \in \text{Pred}_s(v)} \frac{\sigma_{s,v}}{\sigma_{s,w}} (1 + \delta_s(w))
\]

\[
\text{Betweenness}(v) = \sum_{s \in V} \delta_s(v)
\]

As demonstrated above, rather than do a nested loop to calculate \( \sigma_{s,t}(v) \) for Betweenness (which costs \( O(n^2) \) for a single node \( s \)), \( \text{pred}_s(v) \) helps in calculating betweenness by propagating \( \delta_s(v) \) (which costs \( O(m) \) for a single node \( s \) where \( m \) is the number of edges in the graph which is small given the network is edge-sparsed)

Therefore, it helps to reduce the time complexity from \( O(n^3) \) to \( O(nm) \)

### 3.2 Sequential Solution

According to the formulation above the sequential version of the algorithm could be drafted as Algorithm 1.

### 3.3 Algorithm Adjustment and Parallel Solution

To make the algorithm fit for Parallelism, the global variable \( \text{Betweenness} \) is divided by nodes as:

\[
\text{Betweenness}(v) = \sum_{s \in V} \text{Betweenness}_s(v)
\]

Therefore, it could perform in a map-reduce manner, where map the calculation of \( \text{Betweenness}_s(v) \) to the whole list of nodes and reduce all the \( \text{Betweenness}_s(v) \) by summing up by keys. With this approach, we eliminate the strong step-wise independence of the previous algorithm and implement it a virtually pure parallel fashion.

In addition, to reduce the memory cost for each thread, the BFS is calibrated into
a layer-base implementation. Therefore, the distances are not needed to be tracked within a thread and this memory could be saved for its Betweenness\(_s(v)\). Moreover, by deploying layer-based BFS, each branch could be further divided into more thread which make it even suitable for parallelism. However, It turns out that the granularity is so small that the overhead of this further parallelism overwhelm the benefit of itself.

**Algorithm 1** Sequential Style Implementation

\begin{algorithm}
0: \( G \leftarrow \) the graph
0: \( \text{Betweenness} \leftarrow \{n : 0|\forall n \in G\} \)
0: \textbf{for} \( s \) in \( G \) \textbf{do}
0: \( \quad \text{pred} \leftarrow \{n : 0|\forall n \in G\}; \text{dist} \leftarrow \{n : -1|\forall n \in G\}; \text{sigma} \leftarrow \{n : 0|\forall n \in G\}; S \leftarrow \[\] \)
0: \( \quad \text{dist}[s] = 0; \text{sigma}[s] = 1 \)
0: \( \quad \text{queue} \leftarrow [s] \)
0: \( \quad \textbf{while} \ \text{queue} \ \textbf{is \ not \ empty} \ \textbf{do} \)
0: \( \quad \quad v = \text{queue}.\text{get}() \)
0: \( \quad \quad S.\text{append}(v) \)
0: \( \quad \quad \textbf{for} \ w \ \textbf{in} \ G.\text{neighbours}(v) \ \textbf{do} \)
0: \( \quad \quad \quad \textbf{if} \ \text{dist}[w] == -1 \ \textbf{then} \)
0: \( \quad \quad \quad \quad \text{dist}[w] = \text{dist}[v] + 1 \)
0: \( \quad \quad \quad \quad \text{queue}.\text{put}(w) \)
0: \( \quad \quad \quad \textbf{end \ if} \)
0: \( \quad \quad \quad \textbf{if} \ \text{dist}[w] == \text{dist}[v] + 1 \ \textbf{then} \)
0: \( \quad \quad \quad \quad \text{sigma}[w]++ = \text{sigma}[v] \)
0: \( \quad \quad \quad \quad \text{pred}[w].\text{append}(v) \)
0: \( \quad \quad \quad \textbf{end \ if} \)
0: \( \quad \quad \textbf{end \ for} \)
0: \( \quad \deltaelta \leftarrow \{n : 0|\forall n \in G\} \)
0: \( \quad \textbf{for} \ w \ \textbf{in} \ \text{reverse}(S) \ \textbf{do} \)
0: \( \quad \quad \textbf{for} \ v \ \textbf{in} \ pred[w] \ \textbf{do} \)
0: \( \quad \quad \quad \deltaelta[v]++ = \text{sigma}[v]/\text{sigma}[w] * (1 + \deltaelta[w]) \)
0: \( \quad \quad \quad \textbf{if} \ w \neq s \ \textbf{then} \)
0: \( \quad \quad \quad \quad \text{Betweenness}[w]++ = \deltaelta[w]/2 \)
\end{algorithm}
4 Evaluation

4.1 Setting

Experiments are performed on Dell G3 with CPU (8th Gen) i7-8750H Hexa-core 2.20GHz and 16GB DDR4 RAM

4.2 Benchmark

The project focuses on undirected and unweighted social network. To be more specific, it would target on the networks listed in SNAP [4]. It should solve the github network [5] where there are 37,700 nodes and 289,003 edges. Each node denotes a github developer who have at least 10 repositories. Between two nodes, an edge exists if there is a mutual follower. According to our observation, this graph is too large for the sequential implementation. Typically, it would take about 4 days to calculate the result. However, the parallel implementation could solve it in 10 hours which is considerably comparable to a professional network analysis library called NetworkX [6] which is based on a python library SciPy [7] on top of NumPy [8] and C++.

In addition, to incur sequential implementation into the comparison, two edge-spars social networks, Simulate1000 and Simulate 2000 are randomly generated for simulation testing.

4.3 Test Result

To assure the correctness of our implementation, we tested different simulation network with a relatively small size \((n = 10, 50, 100, 500)\) against the NetworkX library [6]. The sum of the differences is lower than \(10^{-13}\). 

<table>
<thead>
<tr>
<th>Graph Name</th>
<th>Number of Nodes</th>
<th>Number of Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulate20</td>
<td>20</td>
<td>96</td>
</tr>
<tr>
<td>Simulate50</td>
<td>50</td>
<td>389</td>
</tr>
<tr>
<td>Simulate100</td>
<td>100</td>
<td>884</td>
</tr>
<tr>
<td>Simulate200</td>
<td>200</td>
<td>1878</td>
</tr>
<tr>
<td>Simulate500</td>
<td>500</td>
<td>4870</td>
</tr>
</tbody>
</table>

For performance, we performed both sequential and parallel version of the algorithm.
with 3 medium and large size graphs.

<table>
<thead>
<tr>
<th>Graph Name</th>
<th>Number of Nodes</th>
<th>Number of Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulate1000</td>
<td>1000</td>
<td>9859</td>
</tr>
<tr>
<td>Simulate2000</td>
<td>2000</td>
<td>19844</td>
</tr>
<tr>
<td>SNAP</td>
<td>37700</td>
<td>289003</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>time(s)</th>
<th>converted</th>
<th>gc'd</th>
<th>fizzled</th>
<th>total</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>6.012</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>3.620</td>
<td>1998</td>
<td>1</td>
<td>1</td>
<td>2000</td>
<td>X1.66</td>
</tr>
<tr>
<td>3</td>
<td>2.590</td>
<td>1998</td>
<td>1</td>
<td>1</td>
<td>2000</td>
<td>X2.32</td>
</tr>
<tr>
<td>4</td>
<td>2.110</td>
<td>1998</td>
<td>1</td>
<td>1</td>
<td>2000</td>
<td>X2.85</td>
</tr>
<tr>
<td>5</td>
<td>1.920</td>
<td>1998</td>
<td>1</td>
<td>1</td>
<td>2000</td>
<td>X3.13</td>
</tr>
<tr>
<td>6</td>
<td>1.780</td>
<td>1998</td>
<td>1</td>
<td>1</td>
<td>2000</td>
<td>X3.38</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>time(s)</th>
<th>converted</th>
<th>gc'd</th>
<th>fizzled</th>
<th>total</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>54.10</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>33.62</td>
<td>3998</td>
<td>1</td>
<td>1</td>
<td>4000</td>
<td>X1.61</td>
</tr>
<tr>
<td>3</td>
<td>27.76</td>
<td>3998</td>
<td>1</td>
<td>1</td>
<td>4000</td>
<td>X2.06</td>
</tr>
<tr>
<td>4</td>
<td>23.36</td>
<td>3998</td>
<td>1</td>
<td>1</td>
<td>4000</td>
<td>X2.32</td>
</tr>
<tr>
<td>5</td>
<td>21.07</td>
<td>3998</td>
<td>1</td>
<td>1</td>
<td>4000</td>
<td>X2.49</td>
</tr>
<tr>
<td>6</td>
<td>20.23</td>
<td>3998</td>
<td>1</td>
<td>1</td>
<td>4000</td>
<td>X2.67</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>44h 43 mins</td>
<td>N/A</td>
</tr>
<tr>
<td>ParBC-N6</td>
<td>18h 38 mins</td>
<td>X2.40</td>
</tr>
<tr>
<td>NetworkX</td>
<td>16h 23 mins</td>
<td>X2.73</td>
</tr>
</tbody>
</table>

### 4.4 Performance Analysis

From the results, we can conclude is much more faster than the sequential version and fairly comparable to NetworkX.

We can also observe that when N = 6, which is equal to the number of cores, the performance of ParBC is the best, about 3.38 times. If N is set to be larger, we can
definitely get better performance since the adjusted version of algorithm can be implemented in a pure parallel manner. Additionally, the computation for each sparks is growing heavier as the size of the graph grows, and when we target at social networks which are inherently large graphs (most of them are even larger than SNAP [4]), the performance improvement of ParBC are guaranteed on those input since the computation heaviness grow faster than parallel overhead.

Figure 1: Eventlog for ParBC-N6 experiment with Simulate1000
5 Future work

During the performance test, we found that our ParBC algorithm reach a bottleneck for performance improvement. We expect better performance speedup as scale of the graph grows, however, it becomes worse when we start to test with Simulate2000 and bigger graphs. After the investigation with heap usage, we found memory size become a major restriction as the figure showed. As the figures suggest, Sequential solution have larger heap space than ParBC-N6 with regarding to available memory per-core wise.

Therefore, for the future work, we will try to investigate and optimize the memory usage with library we covered in the lecture and all other memory-optimization libraries to deal with the obstacle. Hopefully, we can get good performance as expected.

6 Conclusion

In conclusion, ParBC provides a parallel betweenness centrality calculation tool. It calibrates and hence makes itself more suitable for parallelism by deploying layer-wise BFS, dividing total BC into independent sub-BC with regards to different source nodes. This considerably speeds up the sequential implementation and even could be comparable to a C-kernel professional benchmark NetworkX [6]. However, we also notice that the memory size and utilization becomes the bottleneck against the performance, which would be further investigated in our future work.
7 Appendix

Listing 1: Main.hs

module Main where

import System.Exit (die)
import System.Environment (getArgs, getProgName)
import qualified Data.Map.Strict as Map
import BCsequential
import BCparallel

main :: IO ()
main = do args <- getArgs
    case args of
        [version, filename] -> do
            contents <- readFile filename
            let inputMap = Map.fromList rawList
                rawList = map transfromSingleLine rawLines
                rawLines = lines contents
                (int, [int])
            case version of
                "sequential" -> do print version
                    print $ length $ bcSolver inputMap
                "parallel"  -> do print version
                    print $ length $ bcSolverPar inputMap
            _ -> die $ "Usage: Choose correct version (sequential/parallel)"
            _ -> do pn <- getProgName
                    die $ "Usage: $pn<version><filename>"

transfromSingleLine :: String -> (Int, [Int])
transfromSingleLine str = (read node, map read neighbors)
    where
        (node:neighbors) = words str

Listing 2: BasicType.hs

module BasicType where

import qualified Data.Map.Strict as Map
{-#
Sample graph presented by adjacency list
-#}
type Graph = Map.Map Int [Int]

sampleG :: Graph
sampleG = Map.fromList [(1,[2]),(2,[1,3]),(3,[2,4]),(4,[3])]

Listing 3: BCsequential.hs

module BCsequential where

import BasicType
import qualified Data.Map.Strict as Map
import qualified Data.Set as Set

sg :: Graph
sg = sampleG

shortestPathMap g = Map.fromList [((s,e),shortestPath g s e)| s <- Map.keys g, e <- Map.keys g, (/=) s e]

shortestPath :: Graph -> Int -> Int -> Int
shortestPath g s e = bfs e g (Set.fromList [s]) [] 0

bigG :: Graph
bigG = sampleG

bfs :: Int -> Graph -> Set.Set Int -> [Int] -> Int -> Int
bfs target g frontier explored depth |
| Set.member target frontier = depth |
| otherwise = bfs target g newFrontier newExplored (depth+1)
where
newFrontier = Set.fromList (concatMap findchild (Set.toList frontier))
newExplored = explored ++ Set.toList frontier
findchild :: Int -> [Int]
fundchild i
calculateSigmaAndSoOn :: Graph \rightarrow \text{Set} \cdot \text{Set} \cdot \text{Int} \rightarrow \text{Set} \cdot \text{Set} \cdot \text{Int} \rightarrow \text{Map} \cdot \text{Map} \cdot \text{Int} \cdot \text{Int} \rightarrow \text{Map} \cdot \text{Map} \cdot \text{Int} \cdot \text{Int} \rightarrow (\text{Map} \cdot \text{Map} \cdot \text{Int} \cdot \text{Int}, \text{Map} \cdot \text{Map} \cdot \text{Int} \cdot \text{Int})
calculateSigmaAndSoOn g \text{ frontier} s \text{ explored} \text{ prede} \text{ sigma} 
| \text{ Set.null} \text{ frontier} = (\text{ prede} , \text{ sigma} , s) 
| \text{ otherwise} = \text{ calculateSigmaAndSoOn} g \text{ newFrontier} \text{ newS} \text{ newExplored} \text{ newPred} \text{ newSigma} 
\text{ where }
\text{ newFrontier} = \text{ Set.fromList} (\text{ concatMap} \text{ findchild} (\text{ Set.toList} \text{ frontier}))
\text{ newS} = s ++ \text{ Set.toList} \text{ frontier}
\text{ newExplored} = \text{ Set.union} \text{ explored} \text{ frontier}
(\text{ newPred} , \text{ newSigma}) = \text{ updatePredSigma} \text{ prede} \text{ sigma} (\text{ Set.toList} \text{ frontier})
\text{ findchild} :: \text{ Int} \rightarrow [\text{ Int}]
\text{ findchild} i 
| \text{ Map.lookup} i g = \text{ Nothing} = \text{ error} "\text{ invalid\_key\_for\_lookup}"
| \text{ otherwise} = \text{ filter} (\lambda \text{ neighbor} \rightarrow \text{ Set.notMember} \text{ neighbor} \text{ newExplored}) \text{ adjList} 
\text{ where} (\text{ Just} \text{ adjList}) = \text{ Map.lookup} i g
updatePredSigma :: \text{ Map} \cdot \text{ Map} \cdot \text{ Int} \cdot [\text{ Int}] \rightarrow \text{ Map} \cdot \text{ Map} \cdot \text{ Int} \cdot \text{ Int} \rightarrow [\text{ Int}]
\text{ updatePredSigma} p1 s1 [] = (p1, s1)
\text{ updatePredSigma} p1 s1 (v:vs) = \text{ updatePredSigma} p1New s1New vs 
\text{ where }
(p1New, s1New) = \text{ updateForSingleParent} \text{ findchild} v p1 s1
updateForSingleParent :: [\text{ Int}] \rightarrow \text{ Map} \cdot \text{ Map} \cdot \text{ Int} \cdot [\text{ Int}] \rightarrow 
\text{ Map} \cdot \text{ Map} \cdot \text{ Int} \cdot \text{ Int} \rightarrow (\text{ Map} \cdot \text{ Map} \cdot \text{ Int} \cdot \text{ Int}, \text{ Map} \cdot \text{ Map} \cdot \text{ Int} \cdot \text{ Int})
updateForSingleParent [] p2 s2 = (p2, s2)
updateForSingleParent (w:ws) p2 s2 = 
\text{ updateForSingleParent} ws p2New s2New 
\text{ where }
p2New = \text{ Map.insertWith} (\lambda w v p2)
s2New = \text{ Map.insertWith} (\lambda w sig s2)
\text{ where} (\text{ Just} \text{ sig}) = \text{ Map.lookup} v s2
accumulateCB cb _ _ _ _ [] = cb
accumulateCB cb start delta prede sigma (w:ws) = accumulateCB cbNew start deltaNew prede sigma ws

where
  (Just deltaW) = Map.lookup w delta
cbNew = if (=) w start then Map.insertWith (+) w deltaW cb else cb
deltaNew = updateDelta predList delta

where
  (Just predList) = Map.lookup w prede
updateDelta [] d = d
updateDelta (v:vs) d = updateDelta vs dNew
where
dNew = Map.insertWith (+) v dValue d
  dValue :: Double
  dValue = (fromIntegral sigmaV) / (fromIntegral sigmaW) * (1.0 + deltaW) :: Double
  (Just sigmaV) = Map.lookup v sigma
  (Just sigmaW) = Map.lookup w sigma

calculatePerNode :: Graph -> [(Int, [Int])] -> [(Int, Int)] -> [(Int, Double)]
calculatePerNode g iniListPred iniListSigma iniListIntDouble node = Map.map (/ 2.0) resultMap
where
  resultMap = accumulateCB (Map.fromList iniListIntDouble) node (Map.fromList iniListIntDouble) prede sigma (reverse s)
  (prede, sigma, s) = calculateSigmaAndSoOn g (Set.fromList [node]) [] Set.empty (Map.fromList iniListPred) (Map.insert node 1 (Map.fromList iniListSigma))

bcSolver :: Graph -> Map.Map Int Double
bcSolver g = foldl (Map.unionWith (+)) Map.empty bcMapList
where
  bcMapList :: [Map.Map Int Double]
  bcMapList = map (calculatePerNode g iniListPred iniListSigma iniListIntDouble) nodelist
  nodelist :: [Int]
nodelist = Map.keys g
iniListPred :: [(Int, [Int])]
iniListPred = map (\x -> (x,[])) nodelist
iniListSigma :: [(Int, Int)]
iniListSigma = map (\x -> (x,0)) nodelist
iniListIntDouble :: [(Int, Double)]
iniListIntDouble = map (\x -> (x,0.0)) nodelist

Listing 4: BCparallel.hs

module BCparallel where

import BCsequential
import BasicType

import qualified Data.Map.Strict as Map
import Control.Parallel.Strategies

myparMap :: (a -> b) -> [a] -> Eval [b]
myparMap _ [] = return []
myparMap f (a:as) = do b <- rpar (f a)
    bs <- myparMap f as
    return (b:bs)

bcSolverPar :: Graph -> Map.Map Int Double
bcSolverPar g = foldl (Map.unionWith (+)) Map.empty bcMapList
    where
        bcMapList = parMap rpar singleSolver nodelist
        bcMapList = runEval $ myparMap singleSolver nodelist
        singleSolver :: Int -> Map.Map Int Double
                    -- solve Be from one node
        singleSolver = calculatePerNode g iniListPred iniListSigma
                        iniListIntDouble
        nodelist :: [Int]
nodelist = Map.keys g
        iniListPred :: [(Int, [Int])]
        iniListPred = map (\x -> (x,[])) nodelist
        iniListSigma :: [(Int, Int)]
        iniListSigma = map (\x -> (x,0)) nodelist
        iniListIntDouble :: [(Int, Double)]
        iniListIntDouble = map (\x -> (x,0.0)) nodelist

Listing 5: random graph generation

import networkx as nx
from collections import defaultdict

g = nx.generators.random_graphs.powerlaw_cluster_graph(2000,10,0.2,
    seed = 0)
c=0
es = defaultdict(list)
for e in g.edges:
s = e[0]
t = e[1]
if s == t:
    continue
es[s].append(str(t))
es[t].append(str(s))
c+=1

for e in es.keys():
    print(str(e) + " " + " " . join(set(es[e])))
print(c)
References


