Parallel Processing Assignment

Heuristics for the Parallel Louvain Algorithm

CS1607- Deepayan Sanyal
CS1617- Sayak Dey

Contents

Chapter 1

Introduction

In the study of complex networks, a network is said to have community structure if the nodes of the network can be easily grouped into (potentially overlapping) sets of nodes such that each set of nodes is densely connected internally. In the particular case of non-overlapping community finding, this implies that the network divides naturally into groups of nodes with dense connections internally and sparser connections between groups. But overlapping communities are also allowed. The more general definition is based on the principle that pairs of nodes are more likely to be connected if they are both members of the same community, and less likely to be connected if they do not share communities.
Community structures are quite common in real networks. Social networks include community groups (the origin of the term, in fact) based on common location, interests, occupation, etc.
Finding an underlying community structure in a network, if it exists, is important for a number of reasons. Communities allow us to create a large scale map of a network since individual communities act like meta-nodes in the network which makes its study easier.
Individual communities also shed light on the function of the system represented by the network since communities often correspond to functional units of the system.
A very important reason that makes communities important is that they often have very different properties than the average properties of the networks. Thus, only concentrating on the average properties usually misses many important and interesting features inside the networks. For example, in a given social network, both gregarious and reticent groups might exists simultaneously.

Chapter 2

Preliminaries

Mathematical Formulation

Let G(V, E, ω) be an undirected weighted graph, where V is the set of vertices, E is the set of edges and ω(.) is a weighting function that maps every edge in E to a non-zero, positive weight. In the input graph, edges that connect a vertex to itself are allowed — i.e.*(i,i)* can be a valid edge. However, multi-edges are not allowed. Let the adjacency list of i be denoted by $\Gamma(i) = \{j | (i,j) \in E\}$. Let k_i denote the weighted degree of vertex *i* — i.e. $k_i = \sum_{j \in \Gamma(i)} \omega(i,j)$. We will use n to denote the number of vertices in G; M to denote the number of edges in the graph; and m to denote the sum of all edge weights — i.e. $m = \frac{1}{2}\sum_{i \in V} k_i$. A community within graph G represents a (possibly empty) subset of V. In practice, we are interested in partitioning the vertex set V into an arbitrary number of disjoint non-empty communities, each with an arbitrary size (> 0 and <= n). We call a community with just one element as a singlet community. We will use *C(i)* to denote the community that contains vertex *i* in a given partitioning of V. We use the term intra-community edge to refer to an edge that connects two vertices of the same community. All other edges are referred to as inter-community edges. Let $E_{i \to C}$ refer to the set of all edges connecting vertex *i* to vertices in community C. And let $e_{i \to C}$ denote the sum of the edge weights for the edges in $E_{i \to C}$.

$$e_{i \to C} = \sum_{(i,j) \in E_{i \to C}} \omega(i,j)$$

Let a_c denote the sum of degrees of all vertices in community C. Thus ,

$$a_c = \sum_{i \in C} k_i$$

Modularity: Let $p = \{C_1, C_2, \ldots, C_k\}$ denote the set of all communities in a given partitioning of the vertex set V in G(V , E , ω), where 1<=k<= n. Consequently, the modularity (denoted by Q) of the partitioning P is given by the following expression

$$Q = \frac{1}{2m}\sum_{i \in V} e_{i \to C} - \sum_{C \in P} \left(\frac{a_c}{2m} \cdot \frac{a_c}{2m}\right)$$

Modularity is not an ideal metric for community detection and issues such as resolution limit have been identified; a few variants of modularity definitions have also been devised. However, the definition continues to be the more widely adopted version in practice, including in the Louvain method, and therefore, we will use that definition for this paper.
Community Detection: The problem of community detection is to compute a partitioning P of communities that maximizes modularity. This problem has been shown to be NP-Complete.

Chapter 3

Louvain Algorithm

In 2008, Blondel et al. presented an algorithm for community detection. The method, called the Louvain method, is a multi-phase, iterative, greedy heuristic capable of producing a hierarchy of communities. The main idea of the algorithm can be summarized as follows: The algorithm has multiple phases, and within each phase it carries out multiple iterations until a convergence criterion is met. At the beginning of the first phase, each vertex is assigned to a separate community. Subsequently, the algorithm progresses from one iteration to another until the net modularity gain becomes negligible (as defined by a predefined threshold). Within each iteration, the algorithm linearly scans the vertices in an arbitrary but predefined order. For every vertex i, all its neighboring communities (i.e., the communities containing i's neighbors) are examined and the modularity gain that would result if i were to move to each of those neighboring communities from its current community is calculated. Once the gains are calculated, the algorithm assigns a neighboring community that would yield the maximum modularity gain, as the new community for *i* (i.e., new *C(i)*), and updates the corresponding data structures that it maintains for the source and target communities. Alternatively, if all gains turn out to be negative, the vertex stays in its current community. An iteration ends once all vertices are linearly scanned in this fashion. Consequently, the modularity is a monotonically increasing function across iterations of a phase. Once the algorithm converges within a phase, it proceeds to the next phase by collapsing all vertices of a community to a single "meta-vertex"; placing an edge from that meta-vertex to itself with an edge weight that is the sum of weights of all the intra-community edges within that community; and placing an edge between two meta-vertices with a weight that is equal to the sum of the weights of all the inter-community edges between the corresponding two communities. The result is a condensed graph G'(V', E', ω'), which then becomes the input to the next phase. Subsequently, multiple phases are carried out until the modularity score converges. Note that each phase represents a coarser level of hierarchy in the community detection process.
At any given iteration, let $\Delta Q_{i\to C(j)}$ denote the modularity gain that would result from moving a vertex *i* from its current community *C(i)* to a different community *C(j)*. This term is given by:

$$\Delta Q_{i\to C(j)} = \frac{e_{i\to C(j)} - e_{i\to C(i)\setminus\{i\}}}{m} + \frac{2.k_i.a_{C(i)\setminus\{i\}} - 2.k_i.a_{C(j)}}{(2m)^2}$$

Consequently, the new community assignment for i at an iteration is determined as follows. For $j \in \Gamma(i) \cup \{i\}$:

$$C(i) = \underset{C(j)}{arg\,max}\ \Delta Q_{i\to C(j)}$$

While no upper bound has been established on the number of iterations or on the number of phases, it should be evident that the algorithm is guaranteed to terminate with the use of a cutoff for the modularity gain (because of the modularity being a monotonically increasing function until termination). In practice, the method needs only tens of iterations and fewer phases to terminate on most real world inputs.

Chapter 4

Parallel Louvain Algorithm

4.1 Challenges

Any attempt at parallelizing the Louvain method should factor in the sequential nature in which the vertices are visited within each iteration and the impact it has on convergence. Visiting the vertices sequentially gives the advantage of working with the latest information available from all the preceding vertices in this greedy procedure. Furthermore, in the serial algorithm, when a vertex computes its new community assignment, it does so with the guarantee that no other part of the community structure is concurrently being altered. These guarantees may not hold in parallel. In other words, if communities are updated in parallel, it could lead to some interesting situations with an impact on the convergence process as described below:
1. Negative Gain scenario: In a parallel implementation of the Louvain algorithm, there can be cases where two different threads operate on two different nodes to generate community assignments which cause an overall decrease in modularity. Lu et al have provided a pictorial example of such a case as also a mathematical formulation of the problem.

2. Swap scenarios: There exists another scenario that could impede the progression of the parallel algorithm toward a solution. Consider a simple example where two vertices are connected by an edge. In the interest of increasing modularity, if the two vertices make a decision to move to each other's community concurrently, then such an update could potentially result in both vertices simply swapping their community assignments without achieving any modularity gain. This could also happen in a more generalized setting, where subsets of vertices between two different communities swap their community assignments, each unaware of the other's intent to also migrate.

3. Local maxima: A parallel algorithm also runs the risk of settling on locally optimal decisions. This could happen even in serial; in parallel such scenarios may arise if a single community gets partitioned into equally weighted sub-communities, in which there is no incentive for any individual vertex to merge with any of the other sub-communities; and yet, if all vertices from each of the sub-communities were to merge together to form a single community the net modularity gain could be positive. Getting stuck in a locally optimal solution, however, can be resolved when the algorithm progresses to subsequent phases.

4.2 Parallel Heuristics

In this section, a few parallel heuristics will be described.

4.2.1 The Generalized Minimum Label Heuristic

In the parallel algorithm, at any given iteration, if a vertex i has multiple neighboring communities yielding the maximum modularity gain, then the community which has the minimum label among them will be selected as i's destination community. This heuristic is useful mainly for getting rid of swap conditions. While swap situations may delay convergence, they can never lead to nontermination of the algorithm due to the use of a minimum required net modularity gain threshold to continue a phase. As for local maxima, a general proof that effects of elimination of local maxima cases progressively as the algorithm progresses is not possible due to the heuristic nature of algorithm. However, many situations, typically get resolved in subsequent phases; this is because the representation of the individual sub-communities as meta-vertices is likely to lead them to merge with one another forming the containing communities eventually in the output.

4.2.2 Coloring Heuristics

The problem with most problems in parallel implementation of the Louvain algorithm is that the structure of the graph is dynamically changing and the threads running in parallel do not have updated information about the changes in the other threads. Thus, the approach adopted by Hu et al was to color the graph and then process the graph on the basis of colors. First vertices with a particular color are processed and then other color sets are processed. The net result of this is to limit parallelism while keeping the adjacent nodes of a vertex being processed static, thus imposing a limit on the dynamic nature of the graph. A distance-k coloring is defined as a set of color assignment to the vertices such that no two vertices distance k-apart have the same color. Thus, a higher value of k imposes a more rigid limitation to the dynamic nature of the graph structure change. However, this is not useful to counter the challenges to parallelising the algorithm. A graph can always be constructed for which a negative gain scenario will always exist for all values of k(less than the number of vertices).
Despite these lack of guarantees for a positive modularity gain between iterations, coloring still could be effective as a heuristic in practice.. The performance trade-off presented by coloring is a potential reduction in the degree of parallelism versus faster convergence to higher modularity.

4.3 Algorithm

Our parallel algorithm has the following major steps:
(1) Coloring preprocessing: Color the input vertices using distance-k coloring. For this paper, we only explore distance-1 coloring.
(2) Phases: Execute phases one at a time as per Algorithm shown below. Within each phase, the algorithm runs multiple iterations, with each iteration performing a parallel sweep of vertices without locks and using the community information available from the previous iteration. If coloring was applied, then the processing of each color set is parallelized internally and the community information from the previous coloring stages is available to make migration decisions in subsequent coloring stages. This is carried on until the modularity gain between successive iterations becomes negligible.
(3) Graph rebuilding: Between two successive phases, the community assignment output of the completed phase is used to construct the input graph for the next phase. This is done by representing all communities of the completed phase as ‘‘vertices’’ and accordingly introducing edges, identical to the manner in which it is done in the serial algorithm.

Algorithm 1. The parallel Louvain algorithm for a single phase. The inputs are a graph ($G(V, E, \omega)$) and an array of size $|V|$ that represents an initial assignment of community for every vertex C_{init}

```
1: procedure PARALLEL LOUVAIN(G(V, E, ω), C_init)
2:     ColorSets ← Coloring(V), where ColorSets represents a color-based partitioning of V.
           ▷ If the coloring step is omitted, then it automatically implies that all vertices belong to the same color set.
3:     Q_curr ← 0
4:     Q_prev ← −∞                ▷ Current & previous modularity
5:     C_curr ← C_init
6:     while true do              ▷ Iterate until modularity gain becomes negligible.
7:       for each V_k ∈ ColorSets do
8:         C_prev ← C_curr
9:         for each i ∈ V_k in parallel do
10:            N_i ← C_prev[i]
11:            for each j ∈ Γ(i) do N_i ← N_i ∪ {C_prev[j]}
12:            target ← arg max_{t∈N_i} ΔQ_{i→t}
13:            if ΔQ_{i→target} > 0 then
14:              C_curr[i] ← target
15:
16:      C_set ← the set of non-empty communities corresponding to C_curr
17:      Q_curr ← Compute modularity as defined by C_set
18:      if |(Q_curr − Q_prev)/Q_prev| < θ then      ▷ θ is a user specified threshold.
19:        break                                    ▷ Phase termination
20:      else
21:          Q_prev ← Q_curr
```

4.3.1 Pseudo code for kernel function

__global__ void louvain(int V, int m_d, int *ki_d, int *adjacency_d, int *weight_d, int *indi_d, int *community_d, int *community_d_prev, int *result_d, int maxc){

for i <- 0 to maxc //maxc is the maximum color number

community_d_prev[threadid] <- community_d[threadid];

if result_d[threadid] == i then do
maxq <- -inf

for each vetex j adjacent to vertex threadid
q <- $\Delta Q_{i \to c(j)}$ //which is calculated as given in paper using a device function

if q > maxq
maxq <- q
maxcj <- j

if maxq > 0 //update vertex i's community assignment to C(j)
community_d[threaded] <- community_d_prev[j]

}

Chapter 5
Implementation Details

Our implementation was done using C++ and CUDA. We have mainly used C. However some C++ STLs have been used for efficient implementation. Our implementation follows the following code:

```
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <list>
#include <utility>
#include <cmath>
#include <set>
#include <map>

using namespace std;

// A structure that represents an undirected weighted graph
typedef struct
{
   int V, E;
   list< pair <int,int> > *adj;    // A dynamic array of adjacency lists
}Graph;

//device function to calculate the delta-Q when a vertex moves to a different community as given in paper
__device__ float deltaQ(int V, int vi, int cj, int m_d, int *ki_d, int *adjacency_d, int *weight_d, int *indi_d, int
*community_d){
   int eicj=0, eici=0, aci=0, acj=0, i, j;
   float delq;
   //calculating eicj and eici as in paper
   for(i=indi_d[vi]; i<indi_d[vi+1]; i++){
          if(community_d[adjacency_d[i]] == cj){
                 eicj += weight_d[adjacency_d[i]];
          }
          if((community_d[adjacency_d[i]] == community_d[vi]) && (adjacency_d[i] != vi)){
                 eici += weight_d[adjacency_d[i]];
          }
   }
   //calculating aci as in paper
   for(j=0; j<V; j++){
          if(j == vi)
                 continue;
          if(community_d[j] == community_d[vi])
                 aci += ki_d[j];
   }
   //calculating acj as in paper
   for(j=0; j<V; j++){
          if(community_d[j] == cj)
                 acj += ki_d[j];
   }
   delq = (float)(eicj - eici)/(float)m_d + (float)(2*ki_d[vi]*aci -2*ki_d[vi]*acj)/(float)(4*(m_d*m_d));
   return delq;
}

//function to run the louvain algorithm in different threads explained in pseudo code section
__global__ void louvain(int V, int m_d, int *ki_d, int *adjacency_d,  int *weight_d, int *indi_d, int *community_d, int
*community_d_prev, int *result_d, int maxc){

          int tid = threadIdx.x;
          int i, j, maxcj;
          float q, maxq;
```

```
        for(i=0; i<=maxc; i++){
                community_d_prev[tid] = community_d[tid];
                __syncthreads();
                if(result_d[tid] == i){
                        maxq = -1000.0;
                        for(j=indi_d[tid]; j<indi_d[tid+1]; j++){
                                q = deltaQ( V, tid, community_d_prev[adjacency_d[j]], m_d, ki_d, adjacency_d,
weight_d, indi_d, community_d_prev);

                                if(q > maxq){
                                        maxq = q;
                                        maxcj = community_d_prev[adjacency_d[j]];
                                }
                                __syncthreads();
                        }
                        if(maxq > 0){
                                community_d[tid] = maxcj;
                                __syncthreads();
                        }
                        __syncthreads();
                }
        }
}

//create a graph with specified number of edges
Graph create_graph(int v){
   Graph g;
   g.V = v;
          g.E = 0;
   g.adj = new list< pair <int,int> >[v];
   return g;
}

//Add an edge to the graph and update the adjacency lists
void addEdge(Graph *g, int u, int v, int wt)
{
   g->adj[u].push_back(make_pair(v, wt));
   g->adj[v].push_back(make_pair(u, wt));  // Note: the graph is undirected
 }

//add directed edge
void saddEdge(Graph *g, int u, int v, int wt)
{
   g->adj[u].push_back(make_pair(v, wt));
 }

//utility function to create original graph from filename
Graph make_graph(char *filename){

  FILE *fp = fopen(filename , "r");
  if(fp == NULL){
    Graph g = create_graph(0);
    return g;
  }
  int vertices,u,v,edges = 0;
  fscanf(fp , "%d" , &vertices);
  //create graph
  Graph g = create_graph(vertices);
  while((fscanf(fp , "%d %d" ,  &u , &v)) == 2){

    edges++;
    addEdge(&g , u-1 , v-1 , 1);
  }
```

```
  g.E = edges;
    return g;
}

// Assigns colors (starting from 0) to all vertices and prints
// the assignment of colors
void greedyColoring(Graph *g, int *result)
{

    // Assign the first color to first vertex
    result[0]  = 0;

    // Initialize remaining V-1 vertices as unassigned
    for (int u = 1; u < g->V; u++)
        result[u] = -1;  // no color is assigned to u

    // A temporary array to store the available colors. True
    // value of available[cr] would mean that the color cr is
    // assigned to one of its adjacent vertices
    bool available[g->V];
    for (int cr = 0; cr < g->V; cr++)
        available[cr] = false;

    // Assign colors to remaining V-1 vertices
    for (int u = 1; u < g->V; u++)
    {
        // Process all adjacent vertices and flag their colors
        // as unavailable
        list< pair <int,int> >::iterator i;
        for (i = g->adj[u].begin(); i != g->adj[u].end(); ++i)
            if (result[i->first] != -1)
                available[result[i->first]] = true;

        // Find the first available color
        int cr;
        for (cr = 0; cr < g->V; cr++)
            if (available[cr] == false)
                break;

        result[u] = cr; // Assign the found color

        // Reset the values back to false for the next iteration
        for (i = g->adj[u].begin(); i != g->adj[u].end(); ++i)
            if (result[i->first] != -1)
                available[result[i->first]] = false;
    }
}

//calculate the Q-value at the end of each iteration as in paper
float calQ(Graph *g, int *ac, int *community, int m){
    int eici=0, sum=0, j;
    float q;
    //traversing the adjacency list
    for(j=0; j<g->V; j++){
            list< pair <int,int> >::iterator it;
            for (it = g->adj[j].begin(); it != g->adj[j].end(); ++it){
                        if(community[it->first] == community[j]){
                                    eici += it->second;
                        }
            }
            sum += ac[j]*ac[j];
    }
```

```
    q = (float)(eici)/(float)(2*m) - (float)(sum)/(float)(4*m*m); //calculate delQiC(j)
    return q;
}
//Create the condensed graph at the end of each iteration
Graph condensed(Graph *g1, int *community){
        //Make a set to get distinct community numbers from the community assignment array
        set<int> dcom (community,community+g1->V);
        Graph g2 = create_graph(dcom.size());
        int id = 0, cnt = 0;
        // do a mapping of distinct communities  to vertex number for next phase
        std::map<int,int> mymap;
        for (set<int>::iterator itt=dcom.begin(); itt!=dcom.end(); ++itt){
                mymap[*itt] = id++;
        }
        //newad[][]  is the adjacency matrix for condensed graph
        int **newad = new int *[dcom.size()];
        for(int i=0; i<dcom.size(); i++){
                newad[i] = new int[dcom.size()];
                for(int j=0; j<dcom.size(); j++){
                        newad[i][j] = 0;
                }
        }
        //condensing the graph by going through all edges and finding intra and inter community edges
        for(int i=0; i<g1->V; i++){
                list< pair <int,int> >::iterator it;
                for (it = g1->adj[i].begin(); it != g1->adj[i].end(); ++it){
                        newad[mymap.find(community[i])->second][mymap.find(community[it->first])->second] +=
it->second;
                }
        }

        //can omit next two lines if needed
        for(int i=0; i<dcom.size(); i++)
                newad[i][i] = newad[i][i]/2;

        for(int i=0; i<dcom.size(); i++){

                for(int j=0; j<dcom.size(); j++){
                        if(newad[i][j] != 0){
                                if(i != j)
                                        cnt++;
                                        saddEdge(&g2, i, j, newad[i][j]);
                                        g2.E++;
                        }
                }
        }
        g2.E -= cnt/2;
        for(int i = 0; i<dcom.size() ; i++) {
                delete[] newad[i];
         }
        delete[] newad;

        return g2;
}


// main function that implements the parallel Louvain algorithm
int main()
{
  char str[200];
  printf("\n Enter filename with path: ");
  scanf("%s", str);
  Graph g1 = make_graph(str);
```

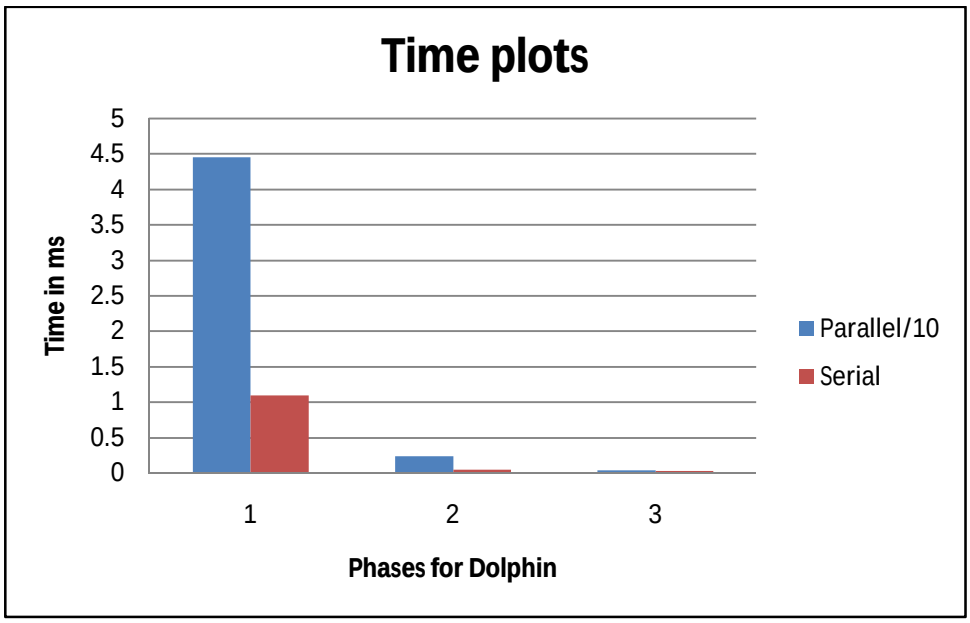
```
  int maxc = -1, m; // maxc max color number
  int result[g1.V]; // color vector
  int indi[g1.V+1]; // index array for adjacency list
  int adjacency[2*g1.E]; // adjacency list
  int weight[2*g1.E]; //weight list
  int ac[g1.V]; //ac as described in paper
  int community[g1.V]; // community assignment array
  int ki[g1.V]; // ki as described in paper
  m = 0; //m as describe in paper
//while number vertices ultimately becomes 1
        while(g1.V > 1){
                //Coloring step
                greedyColoring(&g1, result);
                maxc = -1;
                for(int i=0; i<g1.V; i++){
                        if(result[i]>maxc)
                                maxc = result[i];
                }
                int k=0;
                //making adjacency list and weight list from the graph structure
                for(int i=0; i<g1.V; i++){
                        indi[i] = k;
                        list< pair <int,int> >::iterator it;
                        for (it = g1.adj[i].begin(); it != g1.adj[i].end(); ++it){
                                adjacency[k++] = it->first;
                                weight[k-1] = it->second;
                        }
                }
                indi[g1.V] = k;

                //printing adjacency list
                for(int i=0; i<g1.V; i++){
                        printf("\n %d -> ",i);
                        for(int j=indi[i]; j<indi[i+1]; j++){
                                printf("%d ",adjacency[j]);
                        }
                }


                m = 0;
                //calculating ki and m as in paper
                for(int i=0; i<g1.V; i++){
                        ki[i] = 0;
                        for(int j=indi[i]; j<indi[i+1]; j++){
                                ki[i] = ki[i] + weight[j];
                        }
                        m += ki[i];
                }
                m = m/2;
                printf("\n\n");
                for(int i=0; i<g1.V; i++)
                        community[i] = i;
                //initializing kernel arrays
                int *ki_d, *adjacency_d, *weight_d, *indi_d, *community_d, *community_d_prev, *result_d;
                float Qn, Qo; // Qn <- Qcurr and Qo <- Qprev
                Qo = -99999.0;
                Qn = 0.0;

                //Allocating memory for device arrays and coping respective elements
                cudaMalloc((void **) &ki_d, sizeof(int)*g1.V);
                cudaMalloc((void **) &adjacency_d, sizeof(int)*2*g1.E);
                cudaMalloc((void **) &weight_d, sizeof(int)*2*g1.E);
                cudaMalloc((void **) &indi_d, sizeof(int)*(g1.V+1));
```

```
                cudaMalloc((void **) &community_d, sizeof(int)*g1.V);
                cudaMalloc((void **) &community_d_prev, sizeof(int)*g1.V);
                cudaMalloc((void **) &result_d, sizeof(int)*g1.V);
                cudaMemcpy(ki_d, ki, sizeof(int)*g1.V, cudaMemcpyHostToDevice);
                cudaMemcpy(indi_d, indi, sizeof(int)*(g1.V+1), cudaMemcpyHostToDevice);
                cudaMemcpy(community_d, community, sizeof(int)*g1.V, cudaMemcpyHostToDevice);
                cudaMemcpy(community_d_prev, community, sizeof(int)*g1.V, cudaMemcpyHostToDevice);
                cudaMemcpy(result_d, result, sizeof(int)*g1.V, cudaMemcpyHostToDevice);
                cudaMemcpy(adjacency_d, adjacency, sizeof(int)*2*g1.E, cudaMemcpyHostToDevice);
                cudaMemcpy(weight_d, weight, sizeof(int)*2*g1.E, cudaMemcpyHostToDevice);

                //Each iteration starts here
                while(1){
                        louvain<<<1,g1.V>>>(g1.V, m, ki_d, adjacency_d, weight_d, indi_d, community_d,
community_d_prev, result_d, maxc);
                        // get community assignment
                        cudaMemcpy(community, community_d, sizeof(int)*g1.V, cudaMemcpyDeviceToHost);
                        //Print community assignments
                        printf("\n\n");
                        for(int i=0; i<g1.V; i++){
                                printf("%d ",community[i]);
                        ac[i] = 0;
                        }

                        for(int i=0; i<g1.V; i++)
                                ac[community[i]] += ki[i];
                        printf("\n\n");
                        Qn = calQ(&g1, ac, community, m); // calculate new Q(modulariy) for graph
                        float dq = fabs((Qn - Qo)/Qo);
                        printf("Q = %f, dq = %f\n", Qn, dq);

                        if(dq < 0.0001 || Qn == 0) //Compare with threshold θ = 10^-4
                                break;
                        else
                                Qo = Qn;
                }
                printf("\n\n");
                //graph condensation explained above
                g1 = condensed(&g1, community);

                cudaFree( ki_d );
                cudaFree( adjacency_d );
                cudaFree( weight_d );
                cudaFree( indi_d );
                cudaFree( community_d );
                cudaFree( community_d_prev );
                cudaFree( result_d );
    }
    return 0;
}
```

However, our implementation has a few limitations. The first is that we have implemented the graph coloring in a sequential manner. The other is that the original implementation used C++ mutex locks for synchronization in their parallel implementation. This proved to be beyond our scope and we implemented the parallelism in a sequential manner.

Chapter 6
Results

We ran the Parallel Louvain algorithm on the Dolphin Network consisting of 62 nodes and 159 edges. In the first phase, the algorithm produced 9 clusters. Then the algorithm on the condensed graph with 9 vertices produces 3 communities. In the 3rd phase, the algorithm produced one single community. The timing for the 1st phase was 44.49ms, 2.38 ms for the 2nd phase and 0.32 ms for the 3rd phase. In comparison the serial Louvain took 1.09ms for the 1st phase, 0.046 ms for the 2nd phase and 0.026 ms for the 3rd phase. However, the fact that the serial algorithm is faster than the parallel version is due to the fact that the number of nodes in the graph is comparatively small and the functions to copy the data from device to host and vice-versa combined with the coloring phases make it hard for the parallel algorithm to compete with the serial one for small number of vertices. Hence, we expect the performance of the parallel algorithm to improve with increase in the number of vertices to ultimately better the serial Louvain runtime.



Chapter 7

Conclusion

Through this project, we have tried to implement the parallelization of the Louvain algorithm through some heuristics which were introduced by Hu et al. However, we would like to extend this project into the future. Our future work would include exploring the possibility of parallelizing the rest of the algorithm which were not done here. We would also like to look at other possible heuristics to deal with the negative gain scenario.