Parallelizing Fast Overlapped Community Search (FOCS) Algorithm

Sreyosi Bhattacharyya, Subhra Mazumdar

Mtech CS II, Indian Statistical Institute Kolkata

December 18, 2017

Abstract

Most of the existing algorithms that detect overlapping communities assume that the communities are denser than their surrounding regions, and falsely identify overlaps as communities. FOCS *(Fast Overlapped Community Search)*, an algorithm , that accounts for local connectedness in order to identify overlapped communities, is faster than all previous know techniques, time complexity being linear in number of edges and nodes. It additionally gains in speed via simultaneous selection of multiple near-best communities rather than merely the best, at each iteration. However FOCS evaluates the network structure sequentially. However, the algorithm has good scope of parallelism since clusters expand or decrease in size depending on the behavior of periphery nodes. In case communities do overlap with each other, the algorithm still allows each of them to be treated as independent entities.

1 Problem Definition

Given an unweighted, undirected graph $G(V, E)$ (assuming simple graph, without any self loops or parallel edges), the problem of community detection is to find family of subgraphs $S = \{S_i | S_i \subset V\}$ such that for any node v_j in a subgraph S_i , it is more connected in the subgraph S_i than in another subgraph S_j. Here, $S_j = (S_k | (v_j \notin S_k) \wedge (S_k \in S))$ is any subgraph in family S not containing node v_j. Each subgraph $S_i \in S$ is a community.

For each node v_j , $\forall j \in \{1, 2, \ldots, |V|\}$, let $S(v_j) = \{S_i | (v_j \in S_i) \wedge (S_i \in S)\}$ be the collection of communities containing node v_j . Further, let $S'(v_j) = S - S(v_j)$ be the collection of communities not containing node v_j. If each node v_j belongs to exactly 1, or no community at all, i.e., $|S(v_j)| \leq 1$, then it is called disjoint clustering, overlapped clustering otherwise. FOCS idenitifies overlapped communities in a given graph.

1.1 Basic definition and formulae

For each node v_j, $\forall j \in \{1, 2, \ldots, |V|\}$, v_j is more connected to any community in $S(v_j)$ than any of the communities in $S'(v_j)$. Consequently, we say, v_j is equally well connected to all the communities in $S(v_j)$. This derives the working principle for FOCS.
Let $N(v_j)$ be the set of neighbors of a node $v_j \in V$. Or,

$$N(v_j) = \{v_k | (v_j, v_k) \in E\} \tag{1}$$

Now, let $N_i(v_j)$ be the within community neighborhood of node v_j defined for community $S_i \in S(v_j)$ as follows:

$$N_i(v_j) = \{v_k | (v_j, v_k) \in E \wedge v_k \in S_i\} \tag{2}$$

FOCS defines connectedness of a node with respect to its community as the ratio of the size of its within community neighborhood to the size of the community minus 1. An individual, thus, is considered to be well connected within its community if it has connections to most of the nodes in the community (apart from itself). The *community connectedness score* $\tilde{\zeta}_j^i$, thus, assigned to each node v_j in each community $S_i \in S$ is,

$$\tilde{\zeta}_j^i = \frac{|N_i(v_j)|}{|S_i| - 1} \tag{3}$$

Further, to ensure that a node in any community has at least K neighbours within the community, Equation (3) has been modified to define *community connectedness* score ζ_j^i as follows :

$$\zeta_j^i = \frac{|N_i(v_j)| - K + 1}{|S_i| - K}, \text{if } |N_i(v_j)| > K, \text{ and 0, otherwise.} \tag{4}$$

If K is assigned a very large value, small but dense communities will be missed out. On the other hand, a very small value for K allows for discovery of sparser large communities and insignificant small communities.
The algorithm also defines neighborhood connectedness score ξ_j^i for a node v_j with respect to its community S_i as the ratio of the size of its within community neighborhood to the size of its (overall) neighborhood.

$$\xi_j^i = \frac{|N_i(v_j)|}{|N(v_j)|} \tag{5}$$

This score emphasizes on the fraction of neighborhood of node v_j that is present within the community S_i . It must be noted that community connectedness score decides the belongingness of a node to its community, whereas the *neighborhood connectedness score* only defines the interest of a node in joining a new community.

1.2 Algorithm : parallel version of FOCS

1 **Preprocessing phase** : From the edge list file of the input graph $G(V, E)$ [1], figure out the neighbours of each of the vertices. Do this parallelly using $|V|$ number of threads , each thread used for computation of one vertex. Also simultaneously maintain a count of the neighbour for each vertex. This becomes the *degree* of the vertex.

2 **Initialize Communities** : The nodes with degree greater than $KCORE$ (input parameter) form the central node of each community. This phase can be done parallelly per vertex, unlike in sequential version where each vertex is tested sequentially. Also include the neighbour of the central node in each cluster as the cluster member forming one community. These neighbouring nodes for community S_i must be added to the list $Added_i$ as well.

3 Identify the $near-to-duplicate$ communities by computing the score $\psi(S_i, S_j)$ for each pair of communities $S_i, S_j, i \neq j$. Note that we can form $\binom{|S|}{2}$ pair of communities and compute the overlap score parallelly for each pair using $\binom{|S|}{2}$ pair of threads and eliminate the community which is less in size if the overlap score exceeds OVL.

4 Now after formation of communities, compute the *community connectedness score* for each vertex $v_j \in V$ with respect to each community S_i denoted by ζ_j^i. The computation is done parallely for each community and under each community, the score can be again computed parallely for each members (i.e. vertices) in that community.

[1]Note that we assume the edge list are maintained in increasing order of vertex id. Hence parallelly the threads can read the file for the desired vertex id

Symbol used	Meaning
$G(V,E)$	Input graph
V	Set of all vertices in the graph $G(V,E)$
E	Set of all edges in the graph $G(V,E)$
$\|V\|$	Cardinality of set V
$N(v_j)$	Neighbours of vertex $v_j \in V$ in graph G
S_i	Community denoted by index i consisting of vertices $v \in V$
S	Set of all communities : $S = \cup_i S_i$, i is the index of community
S^k	Set of all communities at iteration k : $S = \cup_i S_i$, i is the index of community k=0 denotes the initialization phase.
$N_i(v_j)$	Neighbours of vertex $v_j \in V$ in community S_i
OVL	Community overlap parameter
$KCORE$	Threshold value for community size
$Added_i$	the added structure for each community S_i
$\psi(S_i, S_j)$	Near-to-duplicate communities for communities S_i and S_j.
ζ_k^i	Community connectedness score for vertex v_k in community S_i
ζ_k^{cutoff}	Stay cut-off score for vertex v_k
ξ_k^i	Neighbourhood connectedness score for vertex v_k in community S_i
ξ_k^{cutoff}	Join cutoff score for vertex v_k
$leave$	Parameter which if set to 1 will causes the peripheral vertices of each community to leave the current community (based on stay cut-off) and decrease the community size
$expand$	Parameter which if set to 1 will causes the peripheral vertices of each community to add their neighbouring vertices(based on join cut-off) and increase the community size

Table 1

5 After computation of *community connectedness score* for all the vertices, then compute the stay cut-off of each vertex parallelly.

6 **Leave Phase** : In this phase, do the computation parallelly for each community. Under each community, do the check of the member vertices of a community parallely. Those members which have community connectedness score less than the stay cutoff , leave the community to which they currently belong.

7 If there exist a community which have size less than $KCORE$, it becomes nonexistent. Else set the global parameter $leave$ to 1 and go back to step [3].

8 Compute the *neighbourhood connectedness score* for each vertex $v_j \in V$ with respect to each community S_i denoted by ξ_j^i. The computation is done parallely for each community and under each community, the score can be again computed parallely for each members (i.e. vertices) in that community.

9 After computation of *neighbourhood connectedness score* for all the vertices, then compute the join cut-off of each vertex parallelly.

10 **Expand Phase** : In this phase, do the computation parallelly for each community. Under each community, do the check of the vertices in set $Added_i$ for a community S_i parallely whether their neighbours not existing in S_i want to join community S_i or not. Those members which have

neighbourhood connectedness score greater than the join cutoff tend to join the community. Add those vertices in set $Nowadded_i$.

11 Set $Added_i$ to $Nowadded_i$. If $|Added_i|$ is greater than equal to 1, then set the parameter *expand* to 1 and go back to step [3] and repeat leave phase.

12 Step [3] to [11] gets repeated till set $Added_i$ becomes ϕ. If $Added_i$ becomes empty, then return the overlapped clusters formed.

Example 1 ***Worked out example on a sample graph by using FOCS-parallel :***
Considering execution of the parallel version of FOCS on the small network given in Figure 1. Parameters considered : OVL is 0.6 and K is 3. Initially 6 communities are formed , which are not disjoint - Cluster : S2, Cluster : S3, Cluster : S5, Cluster : S7, Cluster : S8 and Cluster : S11.

On figuring out near-to-duplicate community structure, Cluster S3 is found to overlap with S2 (intersection being vertices v_1, v_2 *and* $v_3, |S2| = 5, |S3| = 4,$ *) by more than 0.6 (Score is 0.75) and since S3 is smaller in size , it gets eliminated. Similarly is the case with S5 and S7 (intersection being vertices* v_5, v_6 *and* $v_7, |S5| = 5, |S7| = 4,$ *) , overlap is more than 0.6 (score is 0.75) . Thus S7 gets eliminated since its smaller in size. So the clusters remaining are S2, S5, S8 and S11. Now computing the community connectedness score for each vertices in S2, S5, S8 and S11.*

- *For community S2 : Vertex* $v2$ *has score 1,* $v1$ *has score 0,* $v3$ *has score 0,* $v4$ *has score 0 and* $v5$ *has score 0. This is because only* $v2$ *has degree 4 rest have degree less than 4.*
- *For community S5 : Vertex* $v5$ *has score 1,* $v2$ *has score 0,* $v4$ *has score 0,* $v6$ *has score 0 and* $v7$ *has score 0. This is because only* $v5$ *has degree 4 rest have degree less than 4.*
- *For community S8 : Vertex* $v8$ *has score 0,* $v1$ *has score 0,* $v3$ *has score 0,* $v9$ *has score 0 and* $v10$ *has score 0.*
- *For community S11 : Vertex* $v11$ *has score 0,* $v7$ *has score 0,* $v12$ *has score 0 and* $v13$ *has score 0.*

The vertices having degree less than or equal to K is assigned a score 0. So in this case only vertex 2 and 5 manage to get a score of 1 in cluster S2 and S5, but for all other nodes , they get a score of 0 with respect to all the clusters. On computing the stay cutoff for all the vertices , cut off score obtained for all nodes is 0.05.

On execution of the leave phase, since only vertices 2 and 5 have score 1 and rest have score 0 (below the cutoff 0.05) so all the vertices get eliminated from the clusters. So nodes surviving are 2 in S2, 5 in S5. But size of S2 and S5 are 1, rest are of size 0. All the clusters get eliminated as they have size less than K. So no overlapped clusters are detected , algorithm terminates just after execution of one leave phase. Expand phase does not execute.

1.3 Explanation of pseudocode used while implementing parallel version of FOCS in CUDA

The driving principle of FOCS is that communities are initiated by the individuals and influenced by their neighbours and neighbouring communities. A node attracts its neighbouring individuals to be part of the community. Those that find enough connectivity may choose to stay. The communities then expand further as the process is iterated by the newly added members.

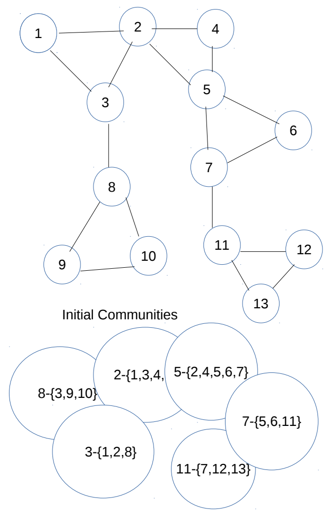


Figure 1: Cluster Formation in a small network

Given the sequential version of the algorithm, we identify the code segment which can be parallelized due to absence of any sort of functional dependency. In the subsequent subsections, we give a detailed description of how all the vertices/clusters can be simultaneously tested for detecting overlap among communities.

1.3.1 Initial Communities

First we need to check the degree of all the nodes existing in the graph $G(V, E)$. Since this is just a read operation based on which we need a count of the neighbouring vertices of the given vertex, this code segment can be executed parallely by $|V|$ number of threads, simultaneously checking the adjacent list assigned for each vertex. Use a shared memory for storing the degree for each vertex. Use this data for forming the intial community structure.

Initially every node $v_i, \forall i \in \{1, 2, \ldots, |V|\}$, that has at least K neighbours, builds a community S_i with its neighbours. Thus the number of initial communities is equal to number of nodes with degree greater than or equal to K. This can also be done parallelly because initial communities comprises the node satisfying the above criteria and its neighbours, allowing overlap between the communities at the initiation (since it may happen one node may be part of more than one cluster). This approach further helps a node participating in multiple communities to selectively stay in more than one community based on high connectedness scores (and leave the rest), simultaneously.

While forming the initial communities, the added structure for each community is also initialized simultaneously, since its just read and store operation, not involving any write operation over shared variable. Let the initial community structure be denoted as S^0 . Further, let $Added_i = \{v_k | v_k \in N(v_i) \wedge k \in S_i\}$, $\forall S_i \in S^0$ be defined and referred to as the set of peripheral nodes of S_i , initially.

The algorithm henceforth iterates over two phases: *leave phase* and *expand phase*. These 2 phases must happen in succession till there exist no more periphery nodes in the *Added* set. Let the community structure obtained after a certain stage l be denoted as S^l.

1.3.2 Duplication Removal

The near duplicate clusters are determined sequentially because of error in synchronization in CUDA over nonexistent communities. At each phase, certain communities may grow to become near-to-duplicate communities. Such near-to-duplicate community pairs (S_i, S_j) are identified via the similarity measure defined as follows:

$$\psi(S_i, S_j) = \frac{|S_i \cap S_j|}{min(|S_i|, |S_j|)} \tag{6}$$

Duplication removal is performed during each stage, before passing communities to leave phase and after every iteration within it. Duplication removal is essential from two viewpoints:

i this prevents the score distribution from being undesirably skewed,

ii with a number of near-to-duplicate communities removed, the computation time is also reduced. Note that this step is done sequentially, due to dependency between pairs of communities and non-existence of one cluster will affect the score computation for other pairs as well.

Duplication removal takes a parameter OVL as input. OVL sets a threshold for the maximum overlap allowed between two communities, before they can be identified as near-duplicates. The smaller of the two communities S_i and S_j is deleted when similarity measure crosses this threshold.

1.3.3 Leave Phase

In this phase a node leaves some of its communities when it finds itself not sufficiently connected in those.The parameter for determining whether a node is sufficiently connected to a community is called community connectedness score as defined in section 1.1 (Equation 4). Thus, a node leaves a community when its community connectedness score falls below the stay cut-off. The method of determining stay cut-off for this parallel version can be found in the later portion of this subsection.

Community connectedness scores are measured in parallel for each community and for each community the score for each vertex is computed sequentially in CUDA due to problem in device to device function call for parallelizing in terms of vertices as well. In a similar fashion we compute the neighbourhood connectedness scores. The connectedness scores are measured using CUDA kernel named as `Cuda_Kernel_Community_Conn_Scores`.

The next step in leave phase is to determine the stay cut-off. Stay cut-off is denoted in this report as $\zeta_i^{cut-off} \forall v_i \in V$.The computation of stay cut-off using community connectedness scores has two parts:
1.Distribution of counts of scores into buckets of size $max(20, N(v_i)$.
2.Computing stay cut-off from the obtained distribution.

In the parallel version,the first step has been done in parallel on all vertices and for each vertex the computation of of score distribution in each bucket is done sequentially.The CUDA kernel named as `Cuda_Kernel_Cutoff _Score` is used for this purpose. Parallelly, stay cut off is also computed for each vertex $v_i \in V$.

.

The remaining portion of the leave phase is done in parallel at cluster level.A cuda kernel named `Cuda_Kernel_Leave_Communities` is used to determine the members $v_k \in cluster_i \forall v_k \in Added_i$ which satisfy the condition $\zeta_k^i < \zeta_k^{cut-off}$. Such vertices are eliminated from a cluster and thus a cluster gets updated. In case an updation of $cluster_i$, we must check whether cardinality of $cluster_i \leq$ KCORE. If this is true then $cluster_i$ is deleted.But if this condition is falls then we set a shared variable called ***leave*** to 1.

1.3.4 Expand Phase

In this phase, a cluster may grow larger by adding some nodes from the periphery. Neighbourhood connectedness scores are measured in parallel for each community and for each community the score for each vertex is computed sequentially. The connectedness scores are measured using CUDA kernel named as `Cuda_Kernel_Neighbourhood_Conn_Scores`. For each community S_i , each adjacent $u \in N(v)$ of each node $v \in Added_i$ is included in S_i if u is not in S_i and it builds up a neighborhood connectedness score greater than its join cut-off. The procedure to measure join cut-off will be discussed later in this subsection.

Join cut-off is denoted in this report as $\xi_i^{cut-off} \forall v_i \in V$.The computation of join cut-off using neighbourhood connectedness scores has two parts:
1.Distribution of counts of scores into buckets
2.Computing join cut-off from the obtained distribution.

In the parallel version,the first step has been computed in parallel on all vertices and for each vertex the computation of distribution is done sequentially.The cuda kernel named as `Cuda_Kernel_Cutoff _Score` is used for this purpose.
The second step is done in parallel on all clusters.

The remaining portion of the expand phase is executed in parallel at cluster level using the cuda kernel `Cuda_Kernel_Expand_Communities`. For each of the remaining communities, we maintain a list to keep a track of the vertices that may get added to a community. To determine the nodes which can be added to a community, we need to check the list of peripheral nodes which have been added to $cluster_i$ in the last level. We check the condition $\xi_k^i > \xi_k^{cut-off}$ and $u_k \notin cluster_i$ for each $u_k \in N(v_j) \forall v_j \in Added_i$ list of peripheral nodes added in last level. If this condition is true then we grow the $cluster_i$ by adding u_k to a new set $NowAdded_i$. If any $cluster_i$ grows by at least one vertex we set a shared variable `expand` to 1. $Added_i$ is set to $NowAdded_i$, to be used in the next phase.

After expand phase, the community structure so obtained at this stage is passed as input to the leave phase of the next stage. At a certain stage, all the peripheral nodes of some particular communities are removed during the leave phase. These communities can not expand further in later stages. However, such a community still contributes to the list of connectedness scores maintained for its nodes. The algorithm stops when, in a stage, there are no peripheral nodes remaining in all existing communities.

1.4 Time complexity analysis

Time taken by the parallel version of the algorithm is greater than the sequential version, due to copying of input data from device to host, host to device or device to device since on execution of CUDA kernel, one needs to create a copy of the input data which is accessible by the kernel. Ignoring this time, the parallel version is faster than sequential as in it simultaneously does computation and number of iteration for convergence will be lesser.

Speed up is achieved in the following cases :

- Counting degree of vertices is done paralllelly instead of sequential computation.
- Formation of communities is done in $\mathcal{O}(1)$ over $\mathcal{O}(n)$, where all vertices are checked parallelly instead of sequentially.
- Computing community connectedness score and neighbourhood connectnedness score are computed parallelly per community wise thus the time complexity reduces to $\mathcal{O}(max(S_i))$ instead of $\mathcal{O}(max(S_i).|S|)$.

- Compute stay cutoff parallelly for each vertex reducing the time complexity to $\mathcal{O}(max(20, |max(N(v_i)|))$ from $\mathcal{O}(|V|.max(20, |max(N(v_i)|))$.
- Compute join cutoff parallelly for each vertex reducing the time complexity to $\mathcal{O}(max(20, |max(N(v_i)|))$ from $\mathcal{O}(|V|.max(20, |max(N(v_i)|))$.
- Leave phase is also computed parallelly, community wise where we check whether member of a community has community connectedness score has value less than stay cut off or not. Time complexity reduces to $\mathcal{O}(max(S_i))$ instead of $\mathcal{O}(max(S_i).|S|)$.
- Expand phase is also computed parallelly, community wise considering the peripheral nodes, where we check whether neighbourhood connectedness score of neighbour of vertices in $Added_i$ has value greater than join cut off or not. Time complexity reduces to $\mathcal{O}(max(S_i))$ instead of $\mathcal{O}(max(S_i).|S|)$.
- Time complexity in total is $\mathcal{O}(max(20, |max(N(v_i)|))$.

2 How to compile the code

For compiling the sequential code in folder $foc\_sequential$:
gcc $-o$ main $graphaslist\_uo\_main.c - lm$
$./main < inputfilename >$ $[(optional) < KCORE >< OVL >]$
Input file is in edge list format where the edges are ordered such that the starting vertices are in ascending order of indices and for a given start vertex, their end vertices must be again in sorted order of indices. Index number of vertex in input file must start from 1.
Output format : Each line represents a community with node labels separated by tab spaces. The values in console output in each line starting with 'phase' represent the change in average number of communities per node across iterations (leave phases).

For compiling the parallel code in folder $foc\_parallel$:
gcc $-o$ main $main\_new.cu$
$./main < inputfilename >$ $[(optional) < KCORE >< OVL >]$
Input file is in edge list format where the edges are ordered such that the starting vertices are in ascending order of indices and for a given start vertex, their end vertices must be again in sorted order of indices. Index number of vertex in input file must start from 1.
Output format : Each line represents a community with node labels separated by tab spaces.

Algorithm 1 Parallel Fast Overlapped Community Search

```
Input               : $G(V,E)$ : input graph, $K$ :minimum connections for a node within a commu-
                      nity, $OVL$: maximum allowed overlap between communities
Output              : $S = \{S_i | S_i \subseteq V$ and $S_i$ is a community$\}$
Auxiliary Variables:  $n = |V|, N(v)$ = neighbours of node $v$,
                      $Added_i$=Nodes added to community $S_i$ in last round
  procedure PREFERRED_COMMUNITIES(G,K,OVL)
      $S = \phi$
      Cuda_Kernel_Find_Neighbour<<< $1, |V|$ >>> $(G, V, |V|)$
      Cuda_Kernel_Initialize_Communities<<< $1, |V|$ >>> $(G, K, S)$
      $expand \leftarrow 1$
      while $expand$ do
        $leave \leftarrow 1$
        while $leave$ do
            $leave \leftarrow 0$
            Detect_Duplicate_Communities$(G, S, OVL)$
            $< \zeta >$=Cuda_Kernel_Community_Conn_Score<<< $1, |S|$ >>> $(G, S, K)$
            $< \zeta^{cutoff} >$=Cuda_Kernel_Cutoff_Score<<< $1, |V|$ >>>(G,S,$< \zeta >$)
            Cuda_Kernel_Leave_Communities<<< $1, |S|$ >>> $S, K, OVL, Leave, < \zeta >, < \zeta^{cutoff} >$
        end
        $expand \leftarrow 0$
        $< \xi >$=Cuda_Kernel_Neighbourhood_Conn_Score<<< $1, |S|$ >>> $(G, S)$
        $< \xi^{cutoff} >$=Cuda_Kernel_Cutoff_Score<<< $1, |V|$ >>>(G,S,$< \xi >$)
        Cuda_Kernel_Expand_Communities<<< $1, |S|$ >>> $S, expand, < \xi >, < \xi^{cutoff} >$
      end
      return $S$
  end procedure

  function CUDA_KERNEL_FIND_NEIGHBOURS$(G, V, |V|)$
  $/ *$ Do in parallel for all vertices $v_i \in V * /$
      Get index of the thread (index $t_i$) which executes on the given vertex $v_i$
      for all neigbours $v_u$ of vertex $v_i$ do
        Increment the count of $N(v_i)$
      end
      return $|N(v_i)|$
  end function
```

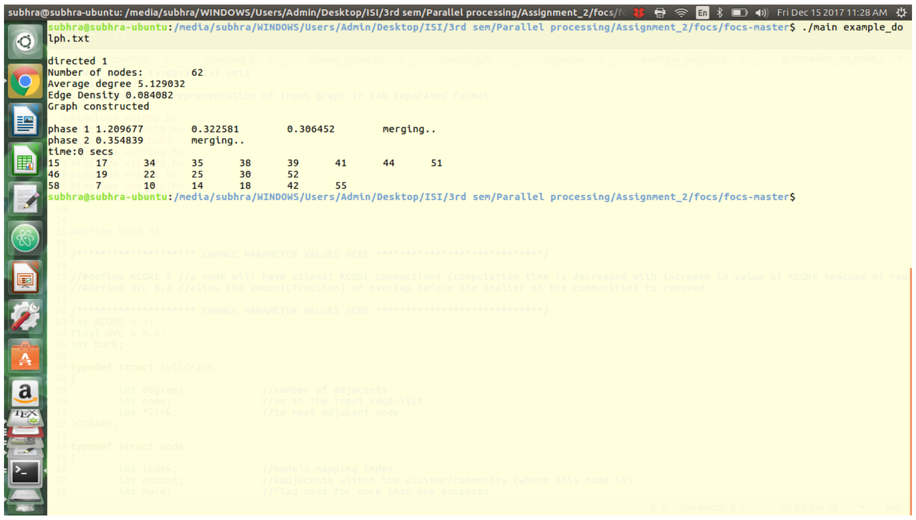
```
function CUDA_KERNEL_INITIALIZE_COMMUNITIES(G,K,S)
/*Community for each node in $G$ initialized by the node $v \in V$
and its neighbors $N(v)$ if $|N(v) \geq K$ */
$/ *$ Do in parallel for all vertices $v_i \in V * /$

    Use a shared variable $S_{temp}$, to be used by all the vertices
    Intialize $S_{temp} = S$
    $syncthreads()$
    Get index of the thread (index $t_i$) which executes on the given vertex $v_i$
    if $|N(v_i) \geq K$ then
        $S_i = \{v_i\} \cup N(v_i)$
        $Added_i \leftarrow N(V_i)$
        $S_{temp} = S_{temp} \cup S_i$
        $syncthreads()$
    end
    else
        $S_i = NULL, , Added_i = NULL$
        $S = S_{temp}$
    end
end function

function DETECT_DUPLICATE_COMMUNITIES(G,S,OVL)
    Eliminate near duplicate community $S_i$ if $\exists u_j \in S_i$ and $i \neq j$
such that $\psi(S_i, S_j) > OVL, \forall S_i \in S$ (as given in Equation 6 )
end function

function CUDA_KERNEL_COMMUNITY_CONN_SCORE(G,S,K)
    Compute community connectedness scores $< \zeta^i >_j$ parallely for
all cluster in $S$ , where each thread $t_i$ deals with one cluster.
    for $v_j \in S_i \wedge v_i \in V$ do
        Compute $\zeta^i_j$ as per Equation (4)
    end
    return $\zeta^i$
end function

function CUDA_KERNEL_NEIGHBOURHOOD_CONN_SCORE(G,S,K)
    Compute neighbourhood connectedness scores $< \xi^i >_j$ parallely for
all cluster in $S$ , where each thread $t_i$ deals with one cluster.
    for $v_j \in S_i \wedge v_i \in V$ do
        Compute $\xi^i_j$ as per Equation (5)
    end
    return $\xi^i$
end function
```

```
function CUDA_KERNEL_CUTOFF_SCORE(G,S,$< \zeta >$) $/ *$ Do in parallel for all vertices $v_i \in V * /$
    Get index of the thread (index $t_i$) which executes on the given vertex $v_i$
    Intialize a bucket of size $max(20, N(v_i))$.
    Set all the values to 0
    for $S_j \in S$ do
        Increment the count in the bucket when a score in the list $\zeta_j^i$
        falls within this range
    end
    • Mark the rightmost bucket having count greater than 0
    • Scan the bucket list from there onwards towards left till a bucket is found that has a count
      lesser than or equal to that of marked bucket and the count of the bucket to its left is greater
      than or equal to that of the current one, or we have reached the leftmost bucket.
    • The lower bound of this bucket is chosen as the stay cut-off for $v_i$.
    • return $\zeta_i^{cutoff}$
end function

function CUDA_KERNEL_LEAVE_COMMUNITIES(S,K,OVL,leave,$< \zeta >$,$< \zeta >^{cutoff}$) $/ *$
Do in parallel for all communities $S_i \in S * /$
    Get index of the thread (index $t_i$) which executes on the each community $S_i$
    $S_i = S_i - \{v_k\}$, if $\zeta_k^{cutoff} > zeta_k^i, \forall v_k \in Added_i$
    if $S_i \leq K$ then
        $S = S - S_i$
    end
    else
        $leave \leftarrow 1$
    end
end function

function CUDA_KERNEL_EXAND_COMMUNITIES(S,expand,$< \xi >$,$< \xi >^{cutoff}$) $/ *$
Do in parallel for all communities $S_i \in S * /$
    Get index of the thread (index $t_i$) which executes on the each community $S_i$
    $Nowadded_i = \phi$
    for $each\ u_k \in N(v_j), \forall v_j \in Added_i$ do
        if $\xi_k^i > \xi_k^{cutoff}$ $and\ u_k \notin S_i$ then
            $S_i = S_i \cup u_k$
            $Nowadded_i = Nowadded_i \cup u_k$
        end
    end
    $Added_i = Nowadded_i$
    if $Added_i \geq 1$ then
        $expand \leftarrow 1$
    end
end function
```

3 Results generated

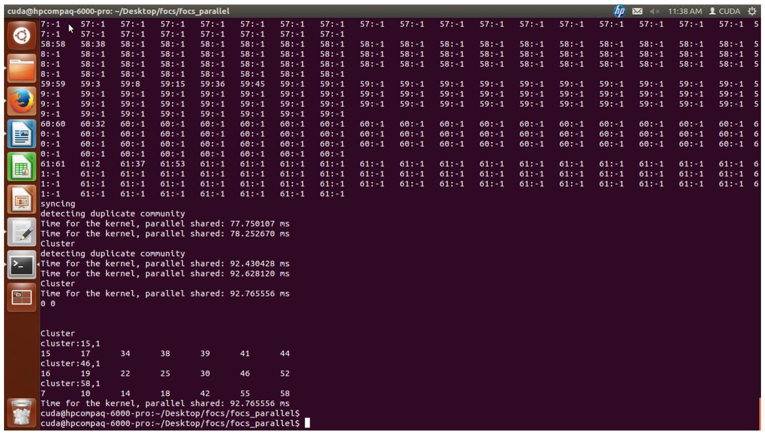
As can be seen from the snapshots of the execution of the code given below for the *DolphinNetwork* having 62 nodes and 159 edges, 5 overlapped clusters are formed in total : $S15, S46$ and $S58$.

The sequential algorithm gives the clusters as $S15, S46$ and $S58$. The result is same as that returned by parallel algorithm.



```
subhra@subhra-ubuntu:/media/subhra/WINDOWS/Users/Admin/Desktop/ISI/3rd sem/Parallel processing/Assignment_2/focs/focs-master$ ./main example_do
lph.txt

directed 1
Number of nodes:        62
Average degree 5.129032
Edge Density 0.084082
Graph constructed

phase 1 1.209677        0.322581        0.306452        merging..
phase 2 0.354839        merging..
time:0 secs
15      17      34      35      38      39      41      44      51
46      19      22      25      30      52
58      7       10      14      18      42      55
subhra@subhra-ubuntu:/media/subhra/WINDOWS/Users/Admin/Desktop/ISI/3rd sem/Parallel processing/Assignment_2/focs/focs-master$
```

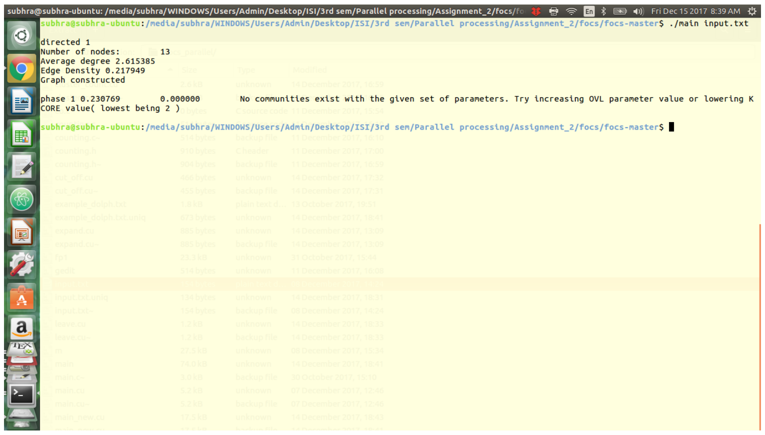
Figure 2: Cluster Formation in a Dolphin network : Sequential Algorithm cluster formation



```
7:-1    57:-1   57:-1   57:-1   57:-1   57:-1   57:-1   57:-1   57:-1   57:-1   57:-1   57:-1   57:-1   57:-1   57:-1   57:-1   57:-1   57:-1   5
7:-1    57:-1   57:-1   57:-1   57:-1   57:-1   57:-1   57:-1
58:58   58:38   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   5
8:-1    58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   5
8:-1    58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   5
8:-1    58:-1   58:-1   58:-1   58:-1   58:-1   58:-1   58:-1
59:59   59:3    59:8    59:15   59:36   59:45   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   5
9:-1    59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   5
9:-1    59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   5
9:-1    59:-1   59:-1   59:-1   59:-1   59:-1   59:-1   59:-1
60:60   60:32   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   6
0:-1    60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   6
0:-1    60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   6
0:-1    60:-1   60:-1   60:-1   60:-1   60:-1   60:-1   60:-1
61:61   61:2    61:37   61:53   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   6
1:-1    61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   6
1:-1    61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   6
1:-1    61:-1   61:-1   61:-1   61:-1   61:-1   61:-1   61:-1
syncing
detecting duplicate community
Time for the kernel, parallel shared: 77.750107 ms
Time for the kernel, parallel shared: 78.252670 ms
Cluster
detecting duplicate community
Time for the kernel, parallel shared: 92.430428 ms
Time for the kernel, parallel shared: 92.628120 ms
Cluster
Time for the kernel, parallel shared: 92.765556 ms
0 0


Cluster
cluster:15,1
15      17      34      38      39      41      44
cluster:46,1
16      19      22      25      30      46      52
cluster:58,1
7       10      14      18      42      55      58
Time for the kernel, parallel shared: 92.765556 ms
cuda@hpcompaq-6000-pro:~/Desktop/focs/focs_parallel$
cuda@hpcompaq-6000-pro:~/Desktop/focs/focs_parallel$
```

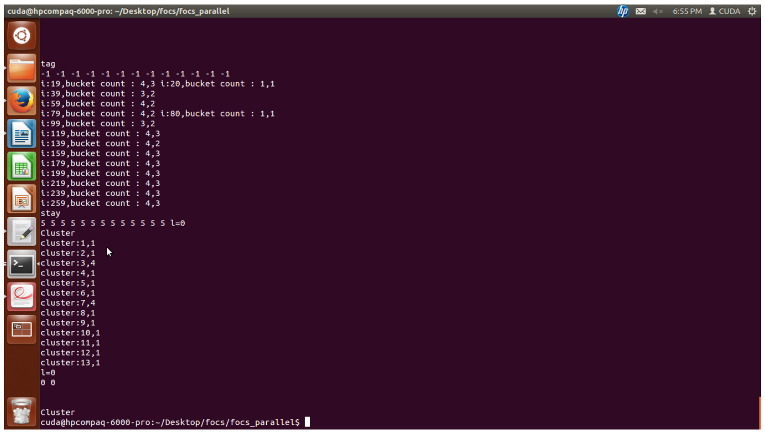
Figure 3: Cluster Formation in a Dolphin network : : Final overlapped cluster set

With reference to network given in Figure 1, the output of the sequential and parallel version of FOCS algorithm is given as well.



```
subhra@subhra-ubuntu:/media/subhra/WINDOWS/Users/Admin/Desktop/ISI/3rd sem/Parallel processing/Assignment_2/focs/focs-master$ ./main input.txt

directed 1
Number of nodes:      13
Average degree 2.615385
Edge Density 0.217949
Graph constructed

phase 1 0.230769        0.000000        No communities exist with the given set of parameters. Try increasing OVL parameter value or lowering K
CORE value( lowest being 2 )

subhra@subhra-ubuntu:/media/subhra/WINDOWS/Users/Admin/Desktop/ISI/3rd sem/Parallel processing/Assignment_2/focs/focs-master$
```

Figure 4: Cluster Formation in a Small network (Figure 1) : Sequential Algorithm cluster formation



```
tag
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
i:19,bucket count : 4,3 i:20,bucket count : 1,1
i:39,bucket count : 3,2
i:59,bucket count : 4,2
i:79,bucket count : 4,2 i:80,bucket count : 1,1
i:99,bucket count : 3,2
i:119,bucket count : 4,3
i:139,bucket count : 4,2
i:159,bucket count : 4,3
i:179,bucket count : 4,3
i:199,bucket count : 4,3
i:219,bucket count : 4,3
i:239,bucket count : 4,3
i:259,bucket count : 4,3
stay
5 5 5 5 5 5 5 5 5 5 5 5 5 l=0
Cluster
cluster:1,1
cluster:2,1
cluster:3,4
cluster:4,1
cluster:5,1
cluster:6,1
cluster:7,4
cluster:8,1
cluster:9,1
cluster:10,1
cluster:11,1
cluster:12,1
cluster:13,1
l=0
0 0


Cluster
cuda@hpcompaq-6000-pro:~/Desktop/focs/focs_parallel$
```

Figure 5: Cluster Formation in a Small Network (Figure 1) : Parallel FOCS

Network	Node	Edges	Cluster formed (FOCS Sequential)	Time taken (Sequential(s))	Cluster formed (FOCS Parallel)	Time taken (FOCS parallel(s))
Small Network	13	17	0	0	0	0.001
Dolphin Network	62	159	3	0	3	0.092
Facebook-ego Network	347	2519	23	0.1	23	11.052

Table 2: Comparison of time taken in detection of communities by FOCS : sequential and parallel

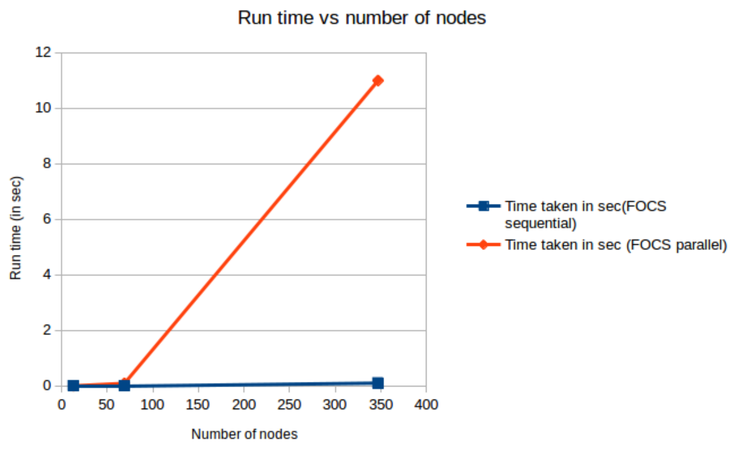


Figure 6: Plot for comparison of time taken in detection of communities by FOCS : sequential and parallel

4 Scope of betterment

In the expand and leave phases we have used cluster-wise parallel computations.We expect that results should be better if within each cluster vertex-wise parallel computations are considered.

5 References for dataset and Sequential version of code

- Youtube, DBLP, Facebook-egonet, LiveJournal, Amazon, Orkut : Stanford Large Network Dataset Collection by Jure Leskovec (https://snap.stanford.edu/data/)
- Dolphin Network, Focs-Sequential source code : (https://github.com/garisha)
- Yeast PPIN, Y2H-Union (http://interactome.dfci.harvard.edu/index.php?page=download)
- Human PPIN, PCDq dataset (http://www.h-invitational.jp/hinv/dataset/download.cgi)

References

[1] S. Bandyopadhyay, G. Chowdhary, and D. Sengupta, “Focs: Fast overlapped community search,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 11, pp. 2974–2985, 2015.

[2] S. Cook, *CUDA Programming:Adeveloper’s Guide to Parallel Computing with GPUs*, 1st ed. Morgan Kaufmann Pubishers INC., 2013.

[3] J. Xie, S. Kelley, and B. K. Szymanski, “Overlapping community detection in networks: The state-of-the-art and comparative study,” *Acm computing surveys (csur)*, vol. 45, no. 4, p. 43, 2013.

[4] A. Lancichinetti, S. Fortunato, and J. Kertész, “Detecting the overlapping and hierarchical community structure in complex networks,” *New Journal of Physics*, vol. 11, no. 3, p. 033015, 2009.

[5] J. Xie and B. K. Szymanski, “Towards linear time overlapping community detection in social networks,” in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2012, pp. 25–36.