HyDetect: A Hybrid CPU-GPU Algorithm for Community Detection

Anwesha Bhowmik
Department of Computational and Data Sciences
Indian Institute of Science
Bangalore, India
banwesha@iisc.ac.in

Sathish Vadhiyar
Department of Computational and Data Sciences
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore, India
vss@iisc.ac.in

Abstract—**Community detection is an important problem that is widely applied for finding cluster patterns in brain, social, biological and many other kinds of networks. In this work, we propose a divide-and-conquer community detection algorithm for hybrid CPU-GPU systems. The graph representing a network is partitioned among the CPU and GPU devices of a node, and independent community detection using Louvain's algorithm is carried out in both the parts. The communities are iteratively refined by a novel strategy for identifying and moving "doubtful" vertices between the devices. The resulting accuracy is found comparable with the single device parallel Louvain algorithms. Our hybrid algorithm helped to explore large graphs that cannot be accommodated in a single device. By harnessing the power of GPUs, our hybrid algorithm is able to provide 42-73% smaller execution times over state-of-art CPU-only algorithms.**

Index Terms—**Community detection, Hybrid CPU-GPU executions, Louvain algorithm.**

I. Introduction

Community detection is an important problem with applications in the analysis of various networks including social, brain and biological networks. A network is modeled as a graph with the entities represented by the vertices and the connections or relations between the entities represented by the edges of the graph. Given a graph, the community detection problem is to find communities in the graph such that the intra-community edges are more than the inter-community edges. Related vertices in the graph are assigned to the same community.

One of the widely used algorithms for community detection is the algorithm by Louvain [1]. This algorithm uses a metric called *modularity* for measuring the goodness of the communities. It iteratively improves the communities of the vertices by moving the vertices to the neighboring communities until the gain in the modularity values due to the movements converge. The algorithm also compresses the graph between the phases such that the communities formed in the previous graph are represented as vertices in the reduced graph. Parallelization of Louvain algorithm is challenging due to high communication and synchronization requirements for calculating the modularities and updating the community information. Recently, parallel Louvain algorithms have been devised for multi-core CPU [2], many-core GPUs [3] and multi-node distributed memory architectures involving CPUs [4], [5]. In this work, we propose a hybrid CPU-GPU algorithm for community detection that simultaneously harnesses both the CPU and GPU resources. The advantage of the hybrid CPU-GPU algorithm is that it can be used to explore large graphs that cannot be accommodated in the GPU memory. By harnessing the power of GPUs, it can also give better performance than the CPU-only algorithms.

Our hybrid algorithm uses the divide-and-conquer paradigm for simultaneous execution on the CPU and GPU cores of a node. The algorithm partitions the graph into two parts, one for the CPU and another for GPU, and simultaneously invokes parallel Louvain algorithm on the respective devices. The communities that are thus formed on the individual devices will be incomplete due to the lack of complete information, i.e., the entire graph. A novel heuristic is employed to determine *doubtful vertices* that have been wrongly assigned to the communities. These doubtful vertices are moved across the devices and Louvain algorithm is executed again on the devices with these doubtful vertices. This process is repeated until the number of communities in the devices converges at which point all the communities are moved to one device and the final set of communities are formed by executing the Louvain algorithm on the device.

Our experiments with large graphs show that the hybrid algorithm gives almost 2x speedup over multi-core parallel Louvain algorithm with less than 1% change in modularity. Our HyDetect algorithm gives at least 42-73% smaller execution times than state-of-art multi-core CPU-only parallel Louvain's algorithm implementation for large graphs that could not be accommodated in the GPU memory.

Section II gives the necessary background, while Section III describes related work in parallel Louvain algorithm. The hybrid CPU-GPU algorithm is explained in Section IV. The implementation details are described in Section V. Section VI presents experiments and results for performance and accuracy. Section VII gives conclusions and future work.

II. Background

A. Modularity

We consider a weighted graph $G(V, E)$ where V is the set of vertices and E is the set of edges. $w_{i,j}$ represents the weight of the edge between vertices i and j. The community detection problem is to partition the graph into a set of communities such

2640-0316/19/$31.00 ©2019 IEEE
DOI 10.1109/HiPC.2019.00013

that the vertices within communities have higher connectivity than vertices of different communities.

Different metrics are used to measure the quality of the communities formed, i.e. to check whether the resulting output has more connectivity within communities than with the other communities. One of the popular metrics for community detection is modularity [6]. Modularity measures the difference between the fraction of edges within the same communities compared to the expected fraction that would exist on a random graph with identical vertex and degree distribution characteristics. The modularity Q of a given graph G can be expressed as

$$Q = 1/2m \sum_{i,j}(A_{i,j} - \frac{k_i * k_j}{2m})\delta(c_i, c_j) \quad (1)$$

where
$A_{i,j}$ represents the adjacency
m=sum of all the edge-weights
k_i= weighted degree of vertex i wrt the edge weights
c_i=community that contains vertex i
$\delta(c_i, c_j) = 1$ if $c_i = c_j$, 0 otherwise.

The modularity depends on the sum of the weights of the edges, denoted as e_c, between the vertices of a community c, and the sum of the weights of all the incident edges on all the vertices of the community c, denoted as a_c. The modularity can also be expressed as

$$Q = \sum_{c \in C}[\frac{e_c}{2m} - (\frac{a_c}{2m})^2] \quad (2)$$

where
$e_c = \sum(w_{i,j}) \; \forall i, j \in c$

B. Louvain's algorithm

One of the commonly used algorithms for community detection is the algorithm by Louvain [1]. Louvain's algorithm consists of multiple phases, and each phase runs in multiple iterations until convergence. The algorithm begins by setting each vertex as a community. In every iteration in a phase, each vertex is moved to a neighborhood community that results in maximum change in modularity. A phase continues until the change in modularity in two successive iterations is less than a threshold value. At the end of a phase, the graph is coarsened such that vertices in a community are collapsed to a new coarse vertex and the sum of the edge weights between two communities is considered as the weight of the edge between the corresponding two coarse vertices. In the next phase, communities are formed in the new coarse graph. This cycle continues until there is not any significant improvement in modularity between two successive phases. Figure 1 illustrates the algorithm. Algorithm 1 shows the steps in a phase of the sequential Louvain algorithm.

Algorithm 1: A Phase of the Sequential Louvain Community Detection

```
1  Input: Graph G=(V,E), threshold $\tau$ ;
2  $Q_{previous} \leftarrow \infty$ ;
3  $C_{previous} \leftarrow$ each vertex itself is a community ;
4  while True do
5      for each $v \in V$ do
6          $N(v) \longleftarrow$ neighbor communities of v ;
7          target= $max_{w \in (N(v))} \Delta Q_{v,w}$ ;
8          if gain then
9              Move v to target and update $C_{current}$ ;
10         end
11     end
12     $Q_{current} \longleftarrow$
           $ModularityCalculation(V, E, C_{current})$ ;
13     if $(Q_{current} - Q_{previous}) < \tau$ then
14         break ;
15     else
16         $Q_{current} \longleftarrow Q_{previous}$ ;
17     end
18 end
```

III. Related Work

Lu et al. proposed a multithreaded version of Louvain algorithm using vertex following, coloring and minimum label heuristic [2]. In the minimum label heuristic, if i and j are two vertices as well as singleton communities, then at the modularity optimization phase (line 7 in Algorithm 1) there is movement from vertex i to vertex j iff c[i]>c[j]. This heuristic helps in preventing conflicting simultaneous movements of vertices to each others' communities. Using graph coloring heuristic, vertices are divided into independent sets and modularity optimization is performed in each set in parallel. Vertex following heuristic discards the isolated vertices as well as one-degree vertices at the first step and then performs Louvain algorithm. This algorithm achieved at least 1.59x speedup over sequential Louvain algorithm in real-world graphs. It achieved 1.4-13.2 speedup over sequential louvain algorithm using Intel Xeon X7560 server with four sockets each with 32 cores and 256 GB of memory. Que et al. [4] proposed a distributed system community detection Louvain algorithm. The algorithm uses efficient strategies to store and process dynamic graphs using hash tables. Fibonacci hashing is used to generate hash functions. Using The hardware systems Zeus, an IBM Power7-IH cluster, and Mira, a 48rack BlueGene/Q system operated by the Argonne Leadership Computing Facility (ALCF) they achieved fair amount of speedup for small,medimum and large real life graphs(upto 49.8x speedup).

Wickramaarachchi et al. [7] presented a coarse-grained algorithm based on MPI for communication. A speedup of about 5x is achieved using 128 processes in this algorithm using 16 compute nodes consisting of two Quad-core AMD Opteron 2376 2.3 GHz processors and 8 cores each node.

Another distributed Louvain algorithm was implemented by Ghosh et al. [5] which partitions the graph randomly. At the end of each iteration communication is involved to obtain information about remote communities. The work reduces the

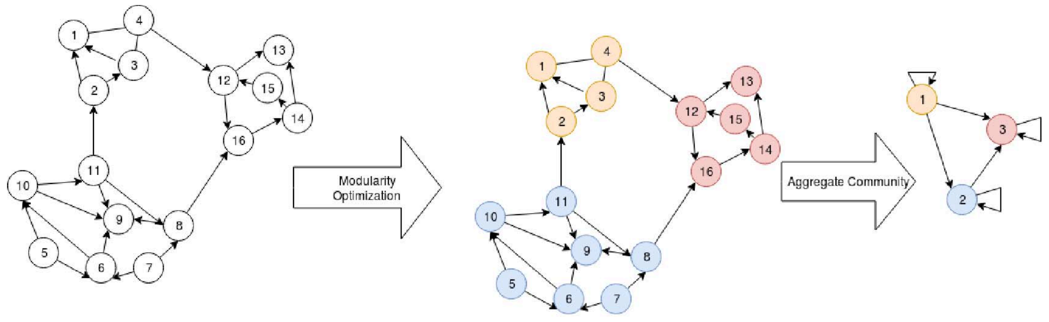


Fig. 1. Louvain Algorithm

communications based on probabilistic heuristics which marks a vertex as active or inactive. An active vertex is a vertex that participates in the next iteration with high probability. Communications and computations are avoided for the inactive vertices. The work achieved 1.8x-46x speedup using NERSC Cori supercomputer which has 32 cores in each node.

Naim et al. [3] presented a GPU based scalable community detection algorithm. The work implements edge-based parallelism. For nodes of highly varying degrees, the nodes are placed in different buckets based on their degrees. Each node is allocated a different number of threads based on its bucket. This achieved 270x speedup compared to sequential implementation using a Tesla K40m GPU with 12 GB of memory, 2880 cores running at 745 MHz, and with CUDA compute capability 3.5 and outperforms existing shared memory implementations.

Cheong et al. [8] presented a hybrid GPU algorithm which uses a coarse grain model across multiple GPUs and a fine grain model in each GPU. The work parallelizes the modularity optimization phase. They obtained speedups in the range of 1.8-5x for single GPU performance and 3-17x when using four GPUs.They used server with Intel Xeon E5405 2 GHz processor and four NVIDIA Fermi C2070 GPUs. However, the work reports low modularity values due to loss of information.

Sharma and Oliveira [9] introduced a novel hybrid community detection strategy which uses both shared and distributed memory. uses a multi-level community detection and coarsening algorithm based on label propagation. They also use a preprocessing step to assign strengths to the edges based on graph topology and to remove weak edges that do not belong to community. Their multi-core shared memory implementation partitions the graph across cores. The cores perform independent assignment of strengths to the edges and removal of weak edges. For label propagation, a global queue is maintained for assigning labels to nodes. The nodes from the queue are cyclically assigned to the cores. This shared memory implementation uses a master thread for synchronization in the strength calculation and label propagation stages. In the distributed memory implementation, the graph is partitioned across nodes. The shared memory algorithm is adopted in each node with inter-process communications during label propagation. The communities are merged in a single master node using a gather operation. Their strategy forms communities with high modularity for large synthetic and real world graphs. It achieved up to 6x speedup using 8 MPI nodes with 16 cores in each node. The use of master threads and master process results in global synchronizations in this algorithm while in our hybrid algorithm, global synchronization is avoided at all levels.

Soman and Narang [10] proposed a variant of the label propagation technique for community detection on multi-core and GPU architectures. In particular, they proposed a weighted label propagation where the edges are assigned weights based on the number of triangles containing the edges. The vertices obtain the labels from the edges. The algorithm has almost linear time complexity and it detects overlapping communities. On multi-core architectures, the algorithm is implemented in a lock-free manner while the GPU implementation uses bitonic sort for the vertices to find the maximum labels in their vicinity. The work demonstrated high speedups on Power 6 architectures with 32 cores and NVIDIA Fermi architectures. Rintu and Vadhiyar [11] developed a generic framework called HyPar for divide-and-conquer executions of graph applications on hybrid CPU-GPU architectures. The work demonstrated the use and performance benefits of the framework with different graph applications including a community detection algorithm that used label propagation. Community detection algorithms using label propagation have high amount of parallelism and provide good speedups at the cost of low modularity values. The Louvain algorithm dealt in this paper has high amount of dependencies and hence difficult to parallelize.

Zeng and Yu [12] have developed a distributed-memory implementation of the Infomap algorithm. Their parallel algorithm duplicates high-degree nodes as delegates to processors for balancing the load among the processors. The Infomap algorithm is a different strategy to Louvain algorithm. It uses a map equation for obtaining a compressed representation of a set of random walks in the graph. The map equation requires the calculation and updates of exit and visit probabilities of the vertices. In a parallel implementation, these updates result in heavy synchronization and communication. The work by Zeng and Yu proposes optimized information swapping strategy among the processors. While their results show linear scalability wrt their algorithm for different processors, the efficiency of the Infomap strategy when compared to Louvain algorithm is yet to be explored.

Our work, while following a hybrid approach also achieves high modularity by using novel techniques for refining the communities. We primarily target large graphs that cannot be accommodated on GPUs.

IV. Hybrid CPU-GPU Community Detection

In our hybrid model, the graph is partitioned into two parts for the two devices, namely, CPU and GPU, of a node. The devices then perform independent and simultaneous Louvain's community detection on their respective parts and form *pseudo communities* based on the partial information corresponding to the parts. These pseudo communities will have vertices that actually do not belong to the communities that would have been determined for the overall graph. Hence the next step in our algorithm is to determine these *doubtful* vertices which are the vertices that do not belong to the communities formed on a device.

The doubtful vertices thus formed in both the devices are then exchanged with the other device. Each device then executes Louvain's algorithm again for the subgraph in the device that includes the communities that were earlier formed and the doubtful vertices that migrated into the device from the other device. This results in the formation of communities with a new set of doubtful vertices. These new doubtful vertices are exchanged again. At each step the graph is coarsened to form a reduced graph of new vertices that correspond to the communities. This continues until the number of communities in each device is small and all the communities of both the devices can be accommodated in a single device. At this stage, all the communities are moved to a device and Louvain's algorithm is executed with all the communities, and the final communities are formed.

The steps of the hybrid algorithm are described in the following subsections.

A. Partitioning

The graph is partitioned into two parts, one for the CPU and the other for the GPU so that the devices can perform independent computations of the Louvain algorithm on their respective parts. The partitioning is based on the proportional performance of the implementations of the Louvain algorithm on the two devices for the given graph. We have explored multiple partitioning strategies including a simple 1-D vertex-block partitioning, Metis[13], [14] and ParMetis[15]. In the 1-D vertex-block partitioning, the CSR (Compressed Sparse Row) arrays representing the graph are divided into two contiguous segments of vertices along with the edges incident on the vertices. As shown in our experiments, we found that the partitioning by Metis gave good performance in most cases.

To determine the ratio of CPU-GPU performance for proportional partitioning based on the performance, a small number of different induced subgraphs (for our study, 3 subgraphs are used) is formed and the implementations of the Louvain's algorithm are executed with each of these subgraphs on both CPU and GPU. The ratio of the execution times on the CPU and GPU is noted and an average of the ratios is obtained for these subgraphs. The original graph is then partitioned into CPU and GPU parts in this ratio by using Metis ratio-based partitioning.

Each subgraph in the above-mentioned procedure is generated randomly such that the number of vertices in the subgraph is 5-10% of the total number of vertices in the original graph. In addition to performance, the GPU memory requirements are also considered to determine the ratio.

B. Independent Computations

After the partitioning, community detection using Louvain's algorithm is performed independently in the CPU and GPU. For independent computations in the CPU, the shared memory multicore algorithm by Lu et al.[2] is used. We extended this algorithm to also calculate internal degree of a vertex in its community and the maximum degree of a vertex in all the communities. These values are used for determining affinity of a vertex to a community in subsequent steps. For GPU, we use the algorithm by Naim et. al[3]. This algorithm allocates different number of threads based on the degree of the vertices.

After the communities are formed in the devices due to the independent computations, the graph is reduced to form a coarse graph with the communities as the vertices and the edges between the communities as the edges between the coarse vertices.

C. Doubtful Vertices

The independent computations in the CPU and GPU results in incomplete or erroneous communities because of the partial information of the graph in the respective parts. A vertex may be wrongly assigned to a community or an isolated vertex may belong to a community.

The vertices that may have been wrongly assigned to communities will have to be determined. These vertices are denoted as *doubtful vertices*. We have developed a novel heuristic for determining the doubtful vertices. We use the concept of *relative commitment* of a vertex in a community. Relative commitment can be defined as the interest of a vertex to be in its community[16]. To find the relative commitment of a vertex, the ratio of the internal degree of the vertex in the community and the maximum internal degree in that community is calculated. The lower this relative commitment of a vertex is, the more prone it is to leave the community.

In addition two more factors are considered to designate vertices as doubtful vertices:

1) A vertex in a community may have a low relative commitment ratio, but it may be connected to the vertices which have a higher ratio in the community. In such cases, the vertices with the lower ratios may be "pulled" into a community by the vertices with the higher ratios.
2) Even if the vertex has a higher ratio in a community, it may be connected to vertices of other communities that have very high ratios. These vertices of other

communities may attract the vertex under consideration into their communities.

Hence, the relative commitment values of both the neighbors of the vertex within the community, denoted as *internal neighbors*, and the neighbors belonging to other communities, denoted as *external neighbors* should be considered.

The relative commitment, RC, of a vertex, v, belonging to a community, c, is calculated as follows.

$$RC(v,c) = \frac{IntDegree(v,c)}{Max_{v_i \in c} IntDegree(v_i,c)} \quad (3)$$

where $IntDegree(v,c)$ is the internal degree of a vertex, v, in a community, c, i.e., the total number of edges connecting the vertex to its neighbors within the community.

The sum of the relative commitments of all the neighbors of v belonging to the same community as v is denoted as $IntRC$.

$$IntRC(v,c) = \sum_{v_i \in neigh(v)} RC(v_i,c)\ \forall v_i \in c \quad (4)$$

Similarly, the sum of the relative commitments of all the neighbors of v belonging to the other communities formed in the same partition in the same device (CPU/GPU) is denoted as $ExtRC$.

The affinity of a vertex, v, to community, c, is then defined as

$$aff(v,c) = \frac{RC(v,c).IntRC(v,c)}{(IntRC(v,c) + ExtRC(v,c))} \quad (5)$$

Note that the affinity values are in the range [0,1]. A vertex, v, is then denoted as a *doubtful vertex* if its affinity, $aff(v,c)$ is less than a threshold, τ.

These doubtful vertices are determined as part of the independent Louvian algorithm calculations in the devices. The calculation of community ids of all vertices and the degree of each vertex in a community are part of independent computations. They are used for determining the doubtful vertices.

The doubtful vertices are isolated from the coarse graph. Thus, at the end of this step, a part in a device will contain the coarse graph plus the isolated doubtful vertices and their connections.

D. Migration of Doubtful Vertices

Some of the vertices in a part on a device are doubtful since they may belong to communities formed in the other device. Hence, the doubtful vertices are candidates for migration to another device. As a next step, the subset of doubtful vertices to be moved to another device will have to be identified.

We use a simple heuristic that calculates the number of border edges and non-border edges of the doubtful vertices. A border edge of a doubtful vertex refers to an edge of the doubtful vertex that resides in the other device. A non-border edges of the doubtful vertex refers to an edge in the same device. We calculate the ratio of the border edges and non border edges of the doubtful vertices. A high ratio implies that the doubtful vertex has more connections to the other device and should be considered for migration to the other device. The algorithm for finding a subset of doubtful vertices for migration is shown in Algorithm 2.

Algorithm 2: Finding Subset of Doubtful Vertices for Migration

```
1  Input: G(V,E),$G_{new}(V_{new}, E_{new})$,threshold $\tau$ ;
2  $MigrationSet = \emptyset$ ;
3  forall $v \in doubtfulVertices$ do
4      $borderedge[v] = 0$ ;
5      $nonborderedge[v] = 0$ ;
6  end
7  forall $v \in doubtfulVertices$ do
8      $borderedge[v] +=$ number of neighbors of $v$ in the
         other device ;
9      $nonborderedge[v] +=$ number of neighbors of $v$ in
         the same device ;
10 end
   /* Second pass                                  */
11 forall $v \in doubtfulVertices$ do
12     if $\frac{borderedge[v]}{nonborderedge[v]} > \tau$ then
13         add $v$ to MigrationSet ;
14     end
15 end
```

The identified doubtful vertices are moved simultaneously between CPU and GPU using cudaMemCpyAsync. At the end of this step, each device will have a coarse graph plus the doubtful vertices in the same device plus the doubtful vertices migrated from the other device.

E. Repeated Invocation of Steps and Obtaining Final Communities

With the communities modified in each device due to isolation and migration of doubtful vertices, the Louvain's algorithm is executed independently on the devices again, the parts are coarsened further, and a new set of doubtful vertices is identified, isolated and migrated. This process is repeated until the reduced graphs in both the parts can be accommodated in a single device, denoted as *finalDevice*. At this point, all the communities and doubtful vertices of both the devices are moved to the $finalDevice$ and Louvain's algorithm is invoked for one final time on the device to obtain the final communities.

In our experiments, it was found that the GPU implementation is at least ten times faster than the CPU implementation. Hence, GPU is chosen as the $finalDevice$.

F. Putting It All Together

The hybrid algorithm is shown in Algorithm 3. The parallelism and simultaneous executions on the CPU and GPU are illustrated in the flowchart in Figure 2.

Algorithm 3: Hybrid Algorithm For Community Detection

```
1 Input: GraphG=(V,E), threshold $\tau$ ;
2 CPU Part,GPU Part $\longleftarrow$ Partition(G,E);
3 repeat
4     $C_{cpu}$,$C_{gpu}$ $\longleftarrow$ independentLouvain(CPU Part,GPU Part) ;
5     $(doubt_{cpu}, doubt_{gpu}) \leftarrow finddoubtful(C_{cpu}, C_{gpu})$ ;
6     $(finaldoubt_{cpu}, finaldoubt_{gpu}) \leftarrow findDoubtfulForMigration(C_{cpu}, C_{gpu})$ ;
7     In parallel move $finaldoubt_{cpu}, finaldoubt_{gpu})$ between CPU and GPU ;
8 until the current coarse graph can be accommodated in GPU;
9 Move all communities to the GPU and perform Louvain Algorithm in the GPU ;
```

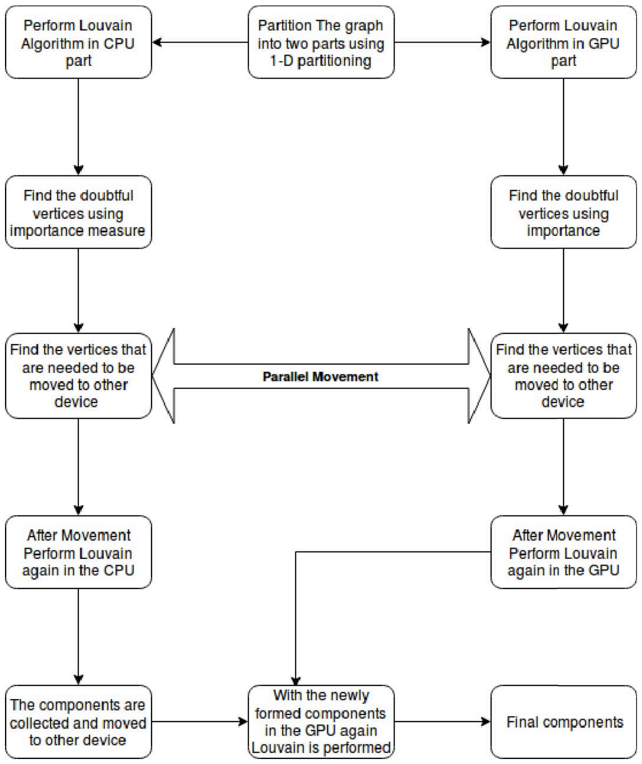

Partition The graph into two parts using 1-D partitioning
Perform Louvain Algorithm in CPU part
Perform Louvain Algorithm in GPU part
Find the doubtful vertices using importance measure
Find the doubtful vertices using importance
Find the vertices that are needed to be moved to other device
Parallel Movement
Find the vertices that are needed to be moved to other device
After Movement Perform Louvain again in the CPU
After Movement Perform Louvain again in the GPU
The components are collected and moved to other device
With the newly formed components in the GPU again Louvain is performed
Final components

Fig. 2. Hybrid Algorithm

V. Implementation and Optimizations

Compressed format is used for storing graph data structures which helps in efficient access to the neighbors of each vertex. The CPU implementation of independent computations uses STL library vector containers and C++ STL map data structure to store the neighboring communities of a vertex. OpenMP[17] is used to achieve parallelism in the independent CPU computation. Boost Graph Library[18], which provides abstractions to access efficient graph data structures, is also used. For migration of the doubtful vertices, the neighbors of each doubtful vertex should be accessed. v in graph g.

For GPU implementation, the Thrust library of CUDA [19] is used. Thrust provides vector containers ($thrust :: host\_vector$ and $thrust :: device\_vector$) similar to STL vector containers. The degree of each vertex in the graph is calculated using $thrust :: transform$ function where the operation is $thrust :: minus$. Minus is a common functor in C++ library.

We also use $thrust :: raw\_pointer\_cast$ to create a raw pointer from a pointer-like type. The device pointer wrapped raw pointer acts like Thrust iterator. It helps to access the device memory from the host. While compressing the graph after a phase, the edge weights of each community need to be updated. For this purpose the reduce function of Thrust is used. It calculates the total edge weights of the same community. For modularity calculations, a sequence of operations needs to be performed. These operations are performed by defining a Thrust user-define functor. A Thrust transform function which uses the user-defined thrust functor can then be used for modularity calculations.

Thrust functor is also defined for doubtful vertex calculations to determine and return if a vertex is doubtful. The functor uses $find$ function that returns the first iterator i in the range $[first, last)$ such that $(*i == value)$ if found or $last$ if no iterator exists. For the doubtful vertex calculations, the $value$ is the maximum relative commitment value which is decided by $thrust :: max\_element$. We also use $reduce\_by\_key$ to add internal degree of the vertices in the same community. Apart from these, $thrust :: fill$ is also used to initialize the $device\_vectors$.

Our software is available for download from https://github.com/marslabiisc/HyDetect.

VI. Experiments and Results

Most of the experiments were performed on a GPU node called *Turing node*. This consists of a 6-core Intel Xeon E5-2620 processor at 2.10 GHz with 24GB RAM and NVIDIA Kepler K40M GPU card. The GPU consists of 2880 streaming processor (SP) cores at 745 MHz and 15 streaming multiprocessors (SMX) with a global memory of 12GB. The compute capability of GPU is 3.5.

Some of the experiments corresponding to the graphs, UK-2002 and RMat24, were performed using a GPU server which consists of a dual octo-core Intel Xeon E5-2670 2.6 GHz processor with Cent OS 6.4, 128 GB RAM, and 1 TB hard disk. The CPU is connected to a NVIDIA Tesla K20m GPU card. The K20m GPU has 2496 GPU cores with a global memory of 4.68 GB at peak memory bandwidth of 208 GB/s.

The graphs used in our experiments are shown in Table I. We primarily consider large graphs that cannot fit within the GPU memory for which our hybrid CPU-GPU work will be useful. The space complexity of the state-of-art GPU implementation[3] is $\mathcal{O}$(3*no of nodes in the graph + 2*no of edges in the graph + maximum degree of the graph). The graphs shown in the table were carefully chosen such that

Graph	$\|V\|$	$\|E\|$	**Approx. Diam.**	**Avg. Deg.**	**Max. Deg.**
uk2002	18.5M	523M	29	28.27	194955
rMat24	16.8M	536M	9	31.9	3582
eu2015	11.2M	759M	8	67.42	398609
gsh	30.8M	1.16B	9	37.73	2.18M
Arabic	22.7M	1.2B	29	55.5	575662
uk2005	39.4M	1.84B	20	46.69	1.78M
it2004	41.2M	2.27B	27	55.01	1.33M

TABLE I
GRAPH SPECIFICATIONS. IN THE TABLE, M STANDS FOR MILLION AND B STANDS FOR BILLION.

their space complexities by the above formula exceed the GPU memory and hence cannot be executed by the state-of-art GPU implementation.

The graphs were obtained from the University of Florida Sparse Matrix Collection [20], the Laboratory for Web Algorithmics [21][22] and the Koblenz Network Collection[23]. As shown in the table, we have used several real world graphs from different categories and having different characteristics including varying degrees for our experiments. These graphs were converted to undirected graphs. GTgraph [24] was used to generate the rMat24 graph. All the results shown are obtained using averages of three runs.

Our hybrid algorithm, referred to as *HyDetect*, is compared with two different implementations: 1. a CPU-only multi-core implementation of Louvain algorithm[2] and 2. a base hybrid CPU-GPU version of Louvain's algorithm. The base hybrid CPU-GPU algorithm partitions the graph into two parts for the CPU and GPU devices, performs independent Louvain algorithm computations on the devices once, moves the communities formed on the CPU to the GPU and executes Louvain's algorithm computations again on the GPU with these communities to form the final communities. The base hybrid version does not involve determination and migration of doubtful vertices. This base hybrid version acts as an intuitive and first-step reasonable implementation of hybrid CPU-GPU algorithm.

A. Partitioning Strategies

We explored three different partitioning strategies, namely, 1-D block partitioning, Metis[13], [14] and ParMetis[15], for the partitioning step of our HyDetect algorithm. The comparisons between the strategies were made in terms of time taken for the partitioning, the number of cut edges produced and the total time taken for our HyDetect algorithm using the resulting partition. This total time also includes the time taken for partitioning. The number of cut edges indicate the quality of the partitioning with a better partitioning strategy expected to yield smaller number of cut edges. The quality of the partitioning can in turn have an impact on the subsequent steps of our HyDetect algorithm in terms of the number of doubtful border vertices and the time taken for migration of these vertices. Table II shows the results.

The 1-D block partitioning strategy consumes only a few seconds for partitioning since it is the simplest strategy that directly divides the CSR arrays. However, the quality of the partitioning is poor as can be seen by the large numer of cut edges produced. Partitioning tools including Metis and ParMetis employ advanced strategies to provide good-quality partitions with small number of cut-edges. Of these, ParMetis takes more time for partitioning due to the parallelization overheads.

In terms of the total times taken by our HyDetect algorithm including the partitioning times, as shown in the third column for each partitioning strategy in the table, we find that employing Metis partitioning results in the least execution times in most cases. Hence, we employ Metis partitioning in our subsequent experiments. The timing results that are shown subsequently include the partitioning times.

B. Thresholds in the Algorithm

Our HyDetect algorithm uses two thresholds: a threshold, P_1, for determining if a vertex is a doubtful vertex and a threshold, $P2$, for deciding if it has to be moved to the other device. We performed experiments with various values of these threshold to analyze the impact of the threshold values on the overall execution times and modularity values. Table III and IV show the results. The best results, in terms of least execution times and highest modularity values, are highlighted in bold.

We find from Table IV that $(P_1 = 0.5, P_2 = 0.6)$ yield the highest modularity values in all cases. We also find from Table IV that these parameter values yield execution times that are competitive with the least execution times. Hence we chose the threshold values as $(P_1 = 0.5, P_2 = 0.6)$ in our subsequent experiments.

C. Comparison with CPU-only and Base Hybrid Methods

Figure 3 compares our hybrid method, HyDetect with the state-of-art multi-core CPU-only and base hybrid methods in terms of execution times and modularities. The modularity values indicate the qualities of the communities formed. As Figure 3(a) shows, HyDetect gives 42-73% lesser execution times than the state-of-art multi-core CPU-only method for comparable modularities. When compared to the base hybrid method, HyDetect gives significantly better modularities for equivalent execution times. This is because the base hybrid method does not detect and isolate the doubtful vertices. Hence, communities are formed in the individual devices with wrongly assigned vertices and these communities are merged in the final step resulting in poor quality communities.

D. Scalability Results

In this section, we analyze the scalability of HyDetect in terms of reduction in execution times over the state-of-art CPU-only algorithm for increasing graph sizes. For this analysis, we add two more large graphs, *Syn-Graph-1* and *Syn-Graph-2*. These are generated using synthetic graph generator suite [24] according to DARPA HPCS SSCA#2 benchmark [25]. Table V shows the results.

We find that our HyDetect algorithm gives significantly lesser execution times than CPU-only algorithm for graphs

Graphs	1-D block			Metis			ParMetis		
	Partitioning Time (s)	Cut Edges (M)	HyDetect Time (s)	Partitioning Time (s)	Cut Edges (M)	HyDetect Time (s)	Partitioning Time (s)	Cut Edges (M)	HyDetect Time (s)
uk-2002	0.78	13.31	10.28	25	1.59	33.85	29	1.57	38.4
rMat24	14	153	936	491	73	1025	803	73	1294
eu-2015	11	116.71	178	71	6.04	201	123	6	500
gsh	19	148.81	906	357	77.56	878	901	77.55	1410
Arabic	12	15.55	163	53.1	2.60	154.1	144	2.71	240

TABLE II
COMPARISON OF DIFFERENT PARTITIONING STRATEGIES IN TERMS OF THE TIME TAKEN FOR PARTITIONING, NUMBER OF CUT EDGES IN MILLIONS (M), TOTAL TIME TAKEN FOR THE HYDETECT ALGORITHM WITH THE PARTITIONING

Graph	$P_1 = 0.2$, $P_2 = 0.3$	$P_1 = 0.35$, $P_2 = 0.4$	$P_1 = 0.5$, $P_2 = 0.6$	$P_1 = 0.65$, $P_2 = 0.7$	$P_1 = 0.7$, $P_2 = 0.8$
uk2002	42	38	33	31	**29**
rMat24	1047	1049	1040	**1021**	1022
eu2015	269	211	251	212	**201**
gsh	800	792	785	785	**784**
Arabic	169	154	151	152	**149**

TABLE III
VARIATION OF EXECUTION TIMES (SECONDS) OF HYDETECT WITH THRESHOLDS

Graph	$P_1 = 0.2$, $P_2 = 0.3$	$P_1 = 0.35$, $P_2 = 0.4$	$P_1 = 0.5$, $P_2 = 0.6$	$P_1 = 0.65$, $P_2 = 0.7$	$P_1 = 0.7$, $P_2 = 0.8$
uk2002	0.96	0.979	0.979	**0.98**	0.971
rMat24	0.44	0.446	**0.459**	**0.45**	**0.451**
eu2015	0.829	0.86	**0.88**	**0.881**	0.879
gsh	0.735	0.74	**0.751**	**0.755**	**0.75**
Arabic	**0.98**	0.96	**0.98**	0.979	**0.98**

TABLE IV
VARIATION OF MODULARITY OF HYDETECT WITH THRESHOLDS

Graph	**Vertices**	**Edges**	**CPU-only time (secs.)**	**HyDetect time (secs.)**	**Exec. Time Reduction**
eu-2015	11.2M	759M	801	451	43
Arabic	22.7M	1.2B	310	154	50%
it-2004	41.2M	2.27B	2105	1123	46%
Syn-Graph-1	41M	3B	2500	1956	21.76%
Syn-Graph-2	49M	3.2B	5970	5910	1.0%

TABLE V
SCALABILITY WITH INCREASING GRAPH SIZES

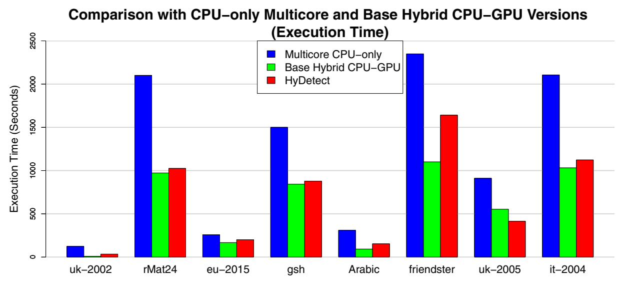
up to 41 million vertices. We obtain about 21% reduction in execution time even for large graphs like *Syn-Graph-1* where only 25% of the graph could be accommodated in the GPU. Our experimental setup and the results show that there is a significant number of large graphs where the GPUs cannot accommodate the graph in entirety but can accommodate at least 25%, for which our hybrid algorithm can be useful and is shown to give smaller execution times than the state-of-art CPU version. It is only for graphs that are larger than this size, i.e., where only less than 25% of the graph can be accommodated in the GPU, our hybrid algorithm gives negligible performance gains over the CPU version.

E. Comparison with State-of-Art

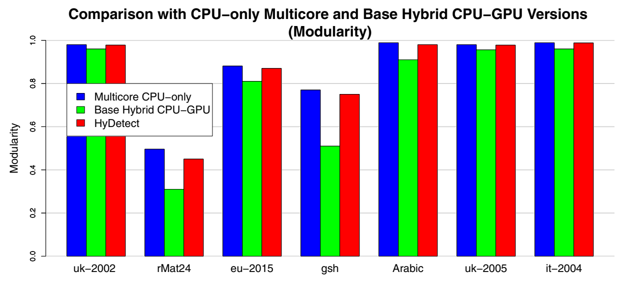
Note that the CPU-only algorithm by Lu et al.[2], over which our HyDetect algorithm shows large improvements, is original state-of-art shared-memory algorithm and one of the highly competent shared-memory algorithms.

The work by Sharma and Oliveira [9] is a distributed memory algorithm. Their work builds on their shared-memory algorithm on a single node and extends to multi-node distributed memory algorithm using MPI+OpenMP. The use of master threads and master process in their algorithm results in global synchronizations in each iteration while in our hybrid algorithm, global synchronization is avoided at all levels. Accordingly, their work shows large times for two graphs. For example, for the Twitter-2010 graph, their algorithm runs in 13000 seconds on 16 cores while our hybrid algorithm, utilizing 6 CPU cores and a K20 GPU, runs in 2101 seconds.

One of the recent works is the distributed Infomap algorithm by Zeng and Yu [12]. The Infomap algorithm is a different strategy to Louvain algorithm. To our knowledge, we are not aware of comparison between the two strategies



(a) Execution Time



(b) Modularity

Fig. 3. Comparison with CPU-only and Base Hybrid Methods

in the literature. In general, Infomap algorithm involves larger communications for exchange of exit and visit probabilities of vertices. For example, for one of the graphs, UK-2005, their algorithm executes in 25 seconds on 512 cores. Assuming linear scalability, which is an optimistic scenario for graph applications, their algorithm will consume 2000+ seconds on 6 cores of CPU. Our HyDetect algorithm, by utilizing 6 CPU cores and an NVIDIA K20 GPU card, is able to execute the same graph in 413 seconds.

F. Execution Phases of HyDetect

Figure 4 shows the times for the different execution phases of HyDetect. We find that most of the times are spent in the useful computations related to independent computations of Louvain's algorithm on the CPU and GPU. These independent computations occupy about 49-77% of the total time. Note that the independent computations also involve identifying the doubtful vertices and finding the subset of doubtful vertices for migration.

Partitioning also takes significant time, about 7-48% of the total time. This is due to the overheads of Metis partitioning employed in our work. However, the high quality Metis partition results in a small number of doubtful vertices. This results in reduced times for migration of the doubtful vertices and the formation of the final communities in the GPU as can be seen in the figure.

We can see that the migration phase, where the final set of doubtful vertices are moved simultaneously between the devices and the Louvain algorithm computations are performed with the already formed communities, takes significantly less

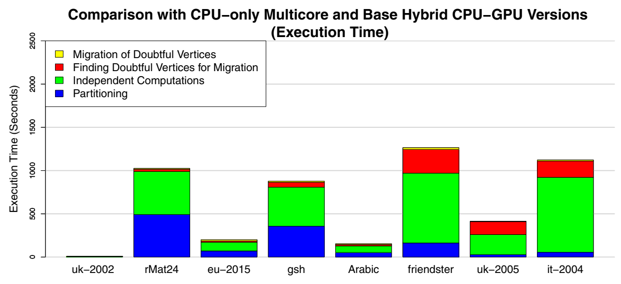


Fig. 4. Times for Various Execution Phases of the HyDetect Algorithm

time compared to initial independent computations. This is due to the reduction in the size of the graph as the algorithm progresses. The formation of the final communities in the GPU consumes only a few seconds.

VII. Conclusion and Future Work

In this work, we have developed an hybrid CPU-GPU community detection algorithm called HyDetect based on Louvain's algorithm. Our hybrid algorithm performs independent and simultaneous Louvain's community detection on the partitions on CPU and GPU, determines wrongly assigned vertices, migrates these vertices in parallel and performs iterative refinement of the communities. We had also developed a novely heuristic for determining wrongly assigned vertices or doubtful vertices. Our experiments show that our HyDetect algorithm gives 42-73% lesser execution times than state-of-art multi-core CPU-only parallel Louvain's algorithm implementation for large graphs that could not be accommodated in the GPU memory. In future, we plan to extend the work to multi-node multi CPU-GPU framework for exploring very large graphs that could not be accommodated in the memory of a single node.

References

[1] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, oct 2008. [Online]. Available: https://doi.org/10.1088%2F1742-5468%2F2008%2F10%2Fp10008

[2] H. Lu, M. Halappanavar, and A. Kalyanaraman, "Parallel heuristics for scalable community detection," *Parallel Computing*, vol. 47, pp. 19 – 37, 2015, graph analysis for scientific discovery. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819115000472

[3] M. Naim, F. Manne, M. Halappanavar, and A. Tumeo, "Community Detection on the GPU," in *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017*, 2017, pp. 625–634.

[4] X. Que, F. Checconi, F. Petrini, and J. Gunnels, "Scalable Community Detection with the Louvain Algorithm," in *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS*, 2015, pp. 28–37.

[5] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, H. Lu, D. Chavarría-Miranda, A. Khan, and A. Gebremedhin, "Distributed Louvain Algorithm for Graph Community Detection," in *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS*, 2018, pp. 885–895.

[6] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E*, vol. 69, p. 026113, Feb 2004. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.69.026113

[7] C. Wickramaarachchi, M. Frincu, P. Small, and V. K. Prasanna, “Fast parallel algorithm for unfolding of communities in large graphs,” in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2014, pp. 1–6.
[8] C. Y. Cheong, H. P. Huynh, D. Lo, and R. S. M. Goh, “Hierarchical parallel algorithm for modularity-based community detection using gpus,” in *Euro-Par 2013 Parallel Processing*, F. Wolf, B. Mohr, and D. an Mey, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 775–787.
[9] R. Sharma and S. Oliveira, “Community detection algorithm for big social networks using hybrid architecture,” *Big Data Res.*, vol. 10, no. C, pp. 44–52, Dec. 2017. [Online]. Available: https://doi.org/10.1016/j.bdr.2017.10.003
[10] J. Soman and A. Narang, “Fast community detection algorithm with gpus and multicore architectures,” in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 568–579. [Online]. Available: https://doi.org/10.1109/IPDPS.2011.61
[11] R. Panja and SVadhiyar, “HyPar: A Divide-and-conquer Model for Hybrid CPU-GPU Graph Processing,” *Journal of Parallel and Distributed Computing*, vol. 132, pp. 8–20, 2019.
[12] J. Zeng and H. Yu, “A Distributed Infomap Algorithm for Scalable and High-Quality Community Detection,” in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018, 2018.
[13] G. Karypis and V. Kumar, “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs,” *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.
[14] ——, “A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices,” *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN*, 1998.
[15] ——, “Parallel Multilevel K-way Partitioning Scheme for Irregular Graphs,” in *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing ’96, 1996.
[16] P. Parau, C. Lemnaru, and R. Potolea, “Assessing vertex relevance based on community detection,” in *2015 7th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K)*, vol. 01, Nov 2015, pp. 46–56.
[17] OpenMP Architecture Review Board, “OpenMP application program interface version 3.0,” May 2008. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf
[18] Boost C++ Libraries. [Online]. Available: http://www.boost.org/
[19] NVIDIA, “Cuda c programming guide version 8.0,” https://docs.nvidia.com/cuda/cuda-c-programming-guide/, September 2016.
[20] T. A. Davis and Y. Hu, “The university of florida sparse matrix collection,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.
[21] P. Boldi and S. Vigna, “The WebGraph framework I: Compression techniques,” in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, 2004, pp. 595–601.
[22] P. Boldi, M. Rosa, M. Santini, and S. Vigna, “Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks,” in *Proceedings of the 20th international conference on World Wide Web*, S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, Eds., 2011, pp. 587–596.
[23] J. Kunegis, “Konect: the koblenz network collection,” in *Proceedings of the 22nd International Conference on World Wide Web*, 2013, pp. 1343–1350.
[24] D. A. Bader and K. Madduri, “Gtgraph: A synthetic graph generator suite,” *Atlanta, GA, February*, 2006.
[25] D. Bader and K. Madduri, “Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors,” in *High Performance Computing - HiPC 2005, 12th International Conference, Goa, India, December 18-21, 2005, Proceedings*, 2005, pp. 465–476.