Received: 17 June 2021 | Revised: 4 January 2022 | Accepted: 18 February 2022
DOI: 10.1002/cpe.6987

RESEARCH ARTICLE

WILEY

Scalable multi-node multi-GPU Louvain community detection algorithm for heterogeneous architectures

Anwesha Bhowmick[1] | **Sathish Vadhiyar**[2] | **Varun PV**[2]

[1]WalMart Labs, Bangalore, India

[2]Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, India

Correspondence
Sathish Vadhiyar, Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, India.
Email: vss@iisc.ac.in

Summary

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 have developed a multi-node multi-GPU Louvain community detection algorithm, simultaneously harnessing the CPU and GPU cores of the devices. The algorithm partitions a given graph across multiple nodes and devices in the nodes and performs independent computations of Louvain algorithm on the parts on the devices. The independently formed communities in the devices are refined by identification of doubtful vertices and migrating them to the other processors. The communities are merged using a hierarchical merging algorithm that ensures that at any point the merged component can be accommodated within a processor. Our experiments show that our algorithm is highly scalable with increasing number of devices and provides large-scale performance for BigData graphs.

KEYWORDS
community detection, GPUs, Louvain algorithm

1 | 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 converges. 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 our previous work,[6] we had proposed a single-node 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.

In this work, we extend our previous work by proposing a distributed memory multi-node multi-device community detection algorithm that follows a divide-and-conquer paradigm. The algorithm first partitions the graph across the compute nodes and further partitions the graph for the CPU and GPU devices in a single node. We invoke Louvain's community detection algorithm on each device for completely independent processing on the devices. The communities that are thus formed on the individual devices will be incomplete due to the lack of complete information, that is,

the entire graph. We had developed a novel heuristic in our previous work to determine *doubtful vertices* that have been wrongly assigned to the communities. We move these doubtful vertices of a processor to another processor. In this work, we have developed heuristics to determine the target processors for migrating the doubtful vertices.

The components of the different processors are then merged using a novel hierarchical merging algorithm. The merging algorithm ensures that at any point, any intermediate merged result formed on a node does not exceed the memory capacity of the node, thus facilitating the exploration of large-scale graphs. Our merging algorithm also harnesses the combined power of both the CPU cores and GPUs on multiple nodes by simultaneous executions on all the devices. The process of independent executions of Louvain algorithm on the devices and hierarchical merging is repeated until the merged component can be accommodated in a single processor. Our experiments show that our multi-node multi-device algorithm is highly scalable for increasing number of devices and GPUs, with 7–8× performance improvement over two-node executions for BigData graphs. Our multi-node algorithm also gives up to 47× speedup over state-of-art single node algorithm and up to 87% performance improvement over state-of-art multi node algorithm.

Section 2 gives the necessary background related to modularity metric and Louvain community detection algorithm. Section 3 describes related work in parallel Louvain algorithm. Section 4 describes our single-node hybrid CPU–GPU algorithm and Section 5 describes our multi-node multi-device algorithm. Section 6 presents experiments and results on scalability and comparison with existing works. Section 7 gives conclusions and future work.

2 | BACKGROUND

2.1 | 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 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, that is, 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.[7] 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

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

where $A_{i,j}$ represents the adjacency; m, sum of all the edge-weights; k_i, weighted degree of vertex i w.r.t. 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

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

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

2.2 | 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. Algorithm 1 shows the steps in a phase of the sequential Louvain algorithm.

Algorithm 1. A phase of the sequential Louvain community detection

Input: Graph G=(V,E), threshold τ

$Q_{previous} \leftarrow \infty$;

$C_{previous} \leftarrow$ each vertex itself is a community ; /* previous community* /

while *True* **do**

  for *each* $v \in V$ **do**

    $N(v) \longleftarrow$ neighbor communities of v ;

    target= $max_{w \in (N(v))} \Delta Q_{v,w}$; /* change in modularity due to movement of vertex *v* to community, *w*. */

    if *gain* **then**

      Move v to target and update $C_{current}$;

    end

  end

  $Q_{current} \longleftarrow ModularityCalculation(V, E, C_{current})$;

  if $(Q_{current} - Q_{previous}) < \tau$ **then**

    break ;

  else

    $Q_{current} \longleftarrow Q_{previous}$;

  end

end

3 | RELATED WORK

There have been many existing works on different algorithms for parallel community detection.

Bae and Howe[8] developed a distributed memory version of the InfoMap algorithm.[9] Zeng and Yu[10] subsequently developed a more scalable version of the 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 work by Zeng and Yu proposes optimized information swapping strategy among the processors.

Moon et al.[11] had developed two parallel versions of the Girvan–Newman (GN) algorithm which uses the concept of edge betweenness to form communities. The first parallel algorithm was developed using Hadoop MapReduce and was demonstrated on Amazon EC2 instances. Palsetia et al.[12] had developed a distributed parallel community detection algorithm based on a novel partitioning strategy. In this strategy, the graph is divided into multiple subdomains and a pair of subdomains is assigned to a processor with duplication of data in the processors. The objective is to reduce the overall communication requirements.

Our work is on the parallelization of widely used Louvain algorithm. We compare our method with parallelization efforts for Louvain community detection algorithm.

3.1 | Parallel Louvain algorithm on multi-core architectures

Fazlali et al.[13] had developed techniques for adaptive parallel thread assignment on multi-core platforms. The thread assignment is for adding neighbor nodes to the community in the Louvain algorithm. The authors had shown that this solution leads to better load balancing among the threads. The number of threads is adapted to the number of neighbors of a vertex and the number of idle cores. The work showed 50% reduction in execution time on 64 cores.

Staudt and Meyerhenke[14] had developed multiple algorithms for shared memory systems including label propagation, Louvain algorithm, and ensemble scheme. Their parallel Louvain algorithm parallelizes both the node migration and coarsening phases. They had also developed an algorithm that extends the Louvain algorithm with a refinement phase for every level after each prolongation. Their ensemble method forms a consensus of base algorithm in the preprocessing step and uses a final algorithm to form the final communities. Using experiments, they identified successful parameters and combinations of the algorithms.

3.2 | GPU-based Louvain algorithm

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. Nodes in the same bucket are accessed in parallel. Two hash tables are used in modularity optimization phase, one for storing incident community identifier and another for storing the weight of the edge from the vertex to the neighboring community. This work achieved 270× speedup compared to sequential implementation using a Tesla K40m GPU and outperformed existing shared memory implementations. This algorithm utilizes GPU efficiently. However, due to the memory limitation it cannot explore large graphs. Our work overcomes this constraint by utilizing both the CPUs and GPUs.

Cheong et al.[15] presented a multi-GPU algorithm which uses a coarse-grained model across multiple GPUs and a fine-grained model within each GPU. The work parallelizes the modularity optimization phase. It partitions the graph in the GPUs and performs Louvain algorithm in each GPU. Then the results from each node are merged using cut edge information. The work obtains speedup in the range of 1.8–5× for single GPU performance and 3–17× when using four GPUs. However, the work reports low modularity values due to the loss of information in partitioning. Our hybrid algorithm provides both accuracy and high performance.

3.3 | Distributed Louvain algorithm

Que et al.[4] proposed a distributed system community detection Louvain algorithm. In each phase as well as in each iteration, the graph structure is changed in the Louvain algorithm. This distributed system algorithm uses efficient strategies to store and process dynamic graphs using hash tables. They use two hash tables: one for incoming edges and another for keeping track of the outgoing edges. One of the main challenges in distributed system Louvain community detection is to maintain convergence properties. In this algorithm, they proposed a novel convergence heuristic based on rigorous experimental analysis. In each iteration, each processor updates its outgoing edge table based on the modularity change till the convergence property is satisfied. Then the graph is reconstructed and the incoming hash table is replaced by the outgoing one for the next phase. The work achieved good speedup and modularity for different real life graphs.

Wickramaarachchi et al.[16] presented a coarse-grained algorithm based on MPI for communication. Most of the time for community detection is spent in the first iteration. Based on this finding, the first iteration of the partitioned graph is performed in the processors independently and for the subsequent iterations the results from each processor are merged. A speedup of about 5× is achieved using 128 processes in this algorithm.

Another distributed Louvain algorithm was implemented by Ghosh et al.[5] which partitions the graph randomly. Each node exchanges its ghost vertices information. Ghost vertices are the vertices which have edges incident with vertices in remote processors. Each node then performs local community detection algorithm and sends information about ghost communities to remote processors. Each processor renumbers its local unique cluster using a map and computes local community number globally using prefix sum. At the end of each iteration, communication is involved to obtain information about remote communities. The work reduces the communication based on probabilistic heuristics which marks a vertex as active or inactive. Communications and computations are avoided for the inactive vertices. The work achieved 1.8–46× speedup.

These distributed algorithms[4,5,16] use only multi-core CPUs. Our work utilizes both CPU and GPU cores in distributed system for fast community detection of large scale graphs.

3.4 | Hybrid CPU–GPU algorithm

The only hybrid CPU–GPU work on community detection, to our knowledge, is the work by Souravlas et al.[17] This work extends their previous work[18] that had developed a parallel community detection algorithm based on threaded binary trees and building paths that result in stronger communities. In the hybrid CPU–GPU work, the CPU first forms the threaded binary trees from the graph data. These trees are then fed to the GPU where they are transformed to a bit-matrix form. The bit-matrix is then transferred to CPU which determines overlapping communities. Thus, in this work the CPU and GPU cores process their respective parallel computations one after the other. However, in our work simultaneous executions are performed on both the CPU and GPU. While this work utilized only a single node, our work is intended for multi-node multi-GPU executions.

4 | SINGLE-NODE HYBRID CPU–GPU LOUVAIN ALGORITHM

In our previous work,[6] we had developed a single-node hybrid CPU–GPU Louvain algorithm. In this 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 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 on both the devices are then exchanged with the other device. Each device then executes Louvain 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, Louvain algorithm is executed with all the communities, and the final communities are formed.

The overall steps are described in the following subsections. More details on the steps, implementation and optimizations can be found in our previous work.[6]

4.1 | Partitioning

For partitioning the graph into two parts, one for CPU and the other for GPU, we use random subgraphs of the graph. We execute CPU only and GPU only Louvain algorithm on the subgraphs. The average ratio of the execution times is used for partitioning the graphs based on proportional performance of the algorithm on the two devices. We have explored multiple partitioning strategies including a simple 1D vertex-block partitioning, Metis[19,20] and ParMetis.[21] Based on our experiments in our previous work,[6] we found that the partitioning by Metis gave good performance in most cases.

4.2 | Independent computations

After the partitioning, community detection using Louvain algorithm is performed independently in the CPU and GPU. For independent computations in the CPU, the shared memory multi-core algorithm by Lu et al.[2] is used with suitable extensions to calculate affinities of the vertices to communities. For GPU, the algorithm by Naim et al.[3] is used.

4.3 | Doubtful vertices

The independent computations in the CPU and GPU result 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*[22] of a vertex in a community. 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, the relative commitment values of 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 affinity of a vertex, v, to a community, c, is then defined as

$$aff(v, c) = \frac{RC(v, c) \times IntRC(v, c)}{(IntRC(v, c) + ExtRC(v, c))}, \tag{3}$$

where

- $RC(v, c)$ is the relative commitment of a vertex v to a community c,
- $IntRC(v, c)$ is the sum of the relative commitments of all the neighbors of v belonging to the same community as v, and
- $ExtRC(v, c)$ 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).

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.

4.4 | Migration of doubtful vertices

Some of the vertices in 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 edge 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 identified doubtful vertices are moved simultaneously between CPU and GPU. 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.

4.5 | 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 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 algorithm is invoked for one final time on the device to obtain the final communities.

5 | MULTI-NODE MULTI-DEVICE LOUVAIN ALGORITHM

In our distributed model, the graph is partitioned across multiple nodes.* Each node maintains border vertices, ghost vertices, and cut edges to the remote processors. Ghost vertices are the vertices belonging to remote processors that are neighbor vertices to the border vertices belonging to a processor. The ghost vertices are maintained in a ghost list.

Similar to the single-node algorithm, independent Louvain algorithm computations are performed on the different nodes and doubtful vertices are identified using relative commitment metric. The doubtful vertices are isolated from the communities formed in the processors. We mark those doubtful vertices that are also border vertices for migration to the other processors.

Unlike the single-node algorithm where a doubtful vertex in a device can potentially be migrated to the other device, in the multi-node algorithm a doubtful vertex identified in a node can potentially be migrated to any one of the other nodes. Hence, the destination node to which the doubtful vertex has to be moved has to be found. We follow a simple heuristic by which a doubtful border vertex is migrated to the neighboring processor to which the doubtful vertex has the maximum number of cut edges. After the movement, the ghost list and the information of the cut edges are updated in the processor.

After the movement, the independent Louvain algorithm computations are performed again on the processors. Doubtful vertices are identified and migrated. This process of independent computations and movement of doubtful vertices is repeated till the number of doubtful vertices does not change significantly between the iterations. At this stage, the communities and doubtful vertices formed in the different processors are considered for merging.

The merging process ensures that the data structures in the different processors considered for merging in a processor can be accommodated in the processor. If the data structures of all the processors can be accommodated in a single processor, merging of the communicates is performed in a processor. Else, a hierarchical merging algorithm is employed. This procedure is continued until the reduced graph can be accommodated in a single node. Once the coarse graph fits in a single node, Louvain community detection is performed on the graph, and the final communities are obtained.

The overall algorithm is illustrated in Figure 1. The steps of the algorithm are described in the following subsections.

5.1 | Partitioning

Our partitioning strategy tries to balance the number of edges in each partition based on the balance degree partitioning used in the Gemini Framework.[23] Initially, the number of edges is divided about equally in the processors. Each MPI process using its rank calculates the read offset and reads the edges from the graph. The processes then use *MPI_Allreduce* to find out the degrees of all the vertices in the graph. Based on the degrees of all the vertices, edge-balanced 1D partitioning across the processors is performed. This is a best attempt strategy to maintain data locality with respect to adjacent vertices within a processor.

After partitioning is done, the cut edge information is maintained using a hash table. Two kinds of vertices are primarily maintained in the hash table, namely, *border vertex* and *ghost vertex*. Border vertices in a partition are those which have cut edges to remote partitions. The adjacent vertices of these border vertices which are in the other partitions are termed as ghost vertices. The cut edges between the border and ghost vertices are denoted as the ghost edges. In the hash table indexed on the processor id, the information about these ghost vertices is stored. After partitioning, each process receives information about the ghost vertices from other processors. The hash table containing all the ghost information is called

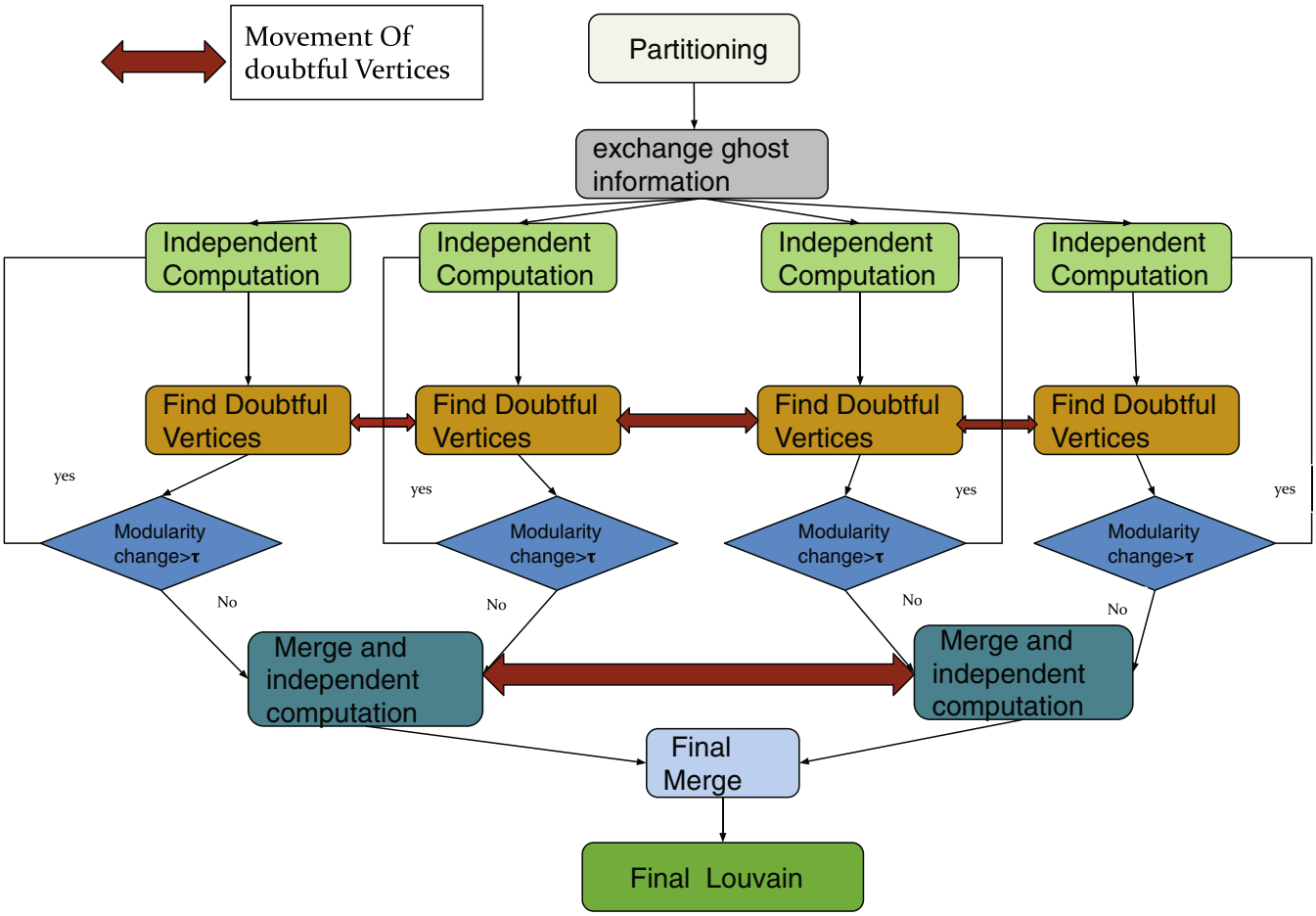

FIGURE 1 Overview of multi-node multi-device Louvain community detection

the *ghostList*. The *ghostList* is exchanged in multiple phases due to the large number of border vertices. The partitioning of a sample graph shown in Figure 2 across four processors is illustrated in Figure 3.

5.2 | Independent computations

After the partitioning, community detection using Louvain algorithm is performed independently in all the nodes on the respective parts of the graph. Based on the size of the graph, there are two options for the independent computations in each part.

1. To execute GPU-only Louvain algorithm in each node.
2. To execute the single node hybrid CPU–GPU HyDetect strategy, described in Section 4, in each node.

If the graph parts are small enough to be accommodated in the GPUs of the nodes, then we execute the GPU-only strategy. Else, *Hydetect* is used for independent computations on the nodes. For the GPU-only strategy, the algorithm proposed by Naim et al.[3] is used.

At the end of the independent computations step, each processor has a coarse subgraph containing the communities formed due to the Louvain algorithm computations.

5.3 | Migration of doubtful vertices

Doubtful vertices in each processor are found by applying the same procedure as in *HyDetect* using relative commitment and affinity of the vertices.

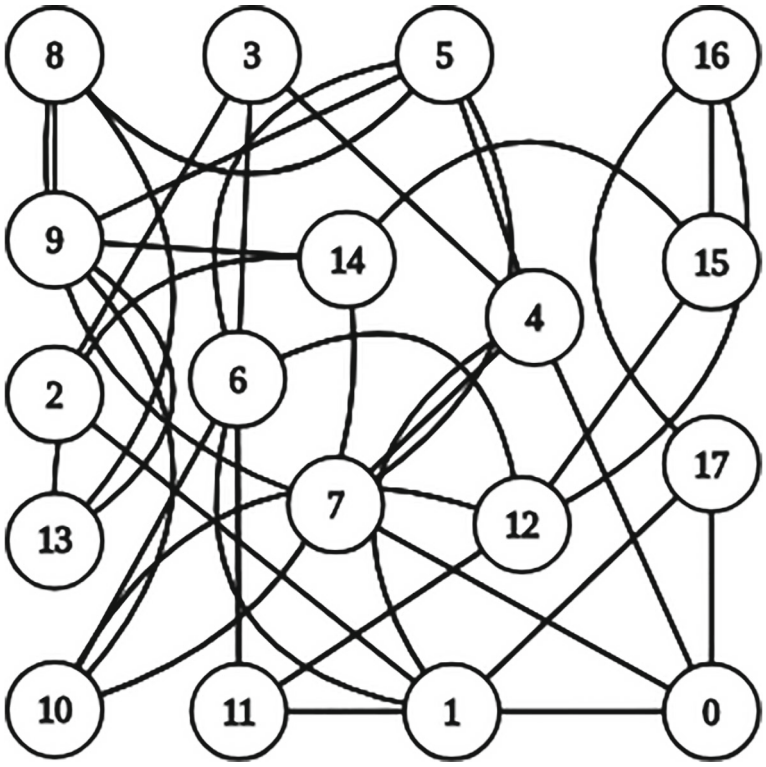

FIGURE 2 An example graph

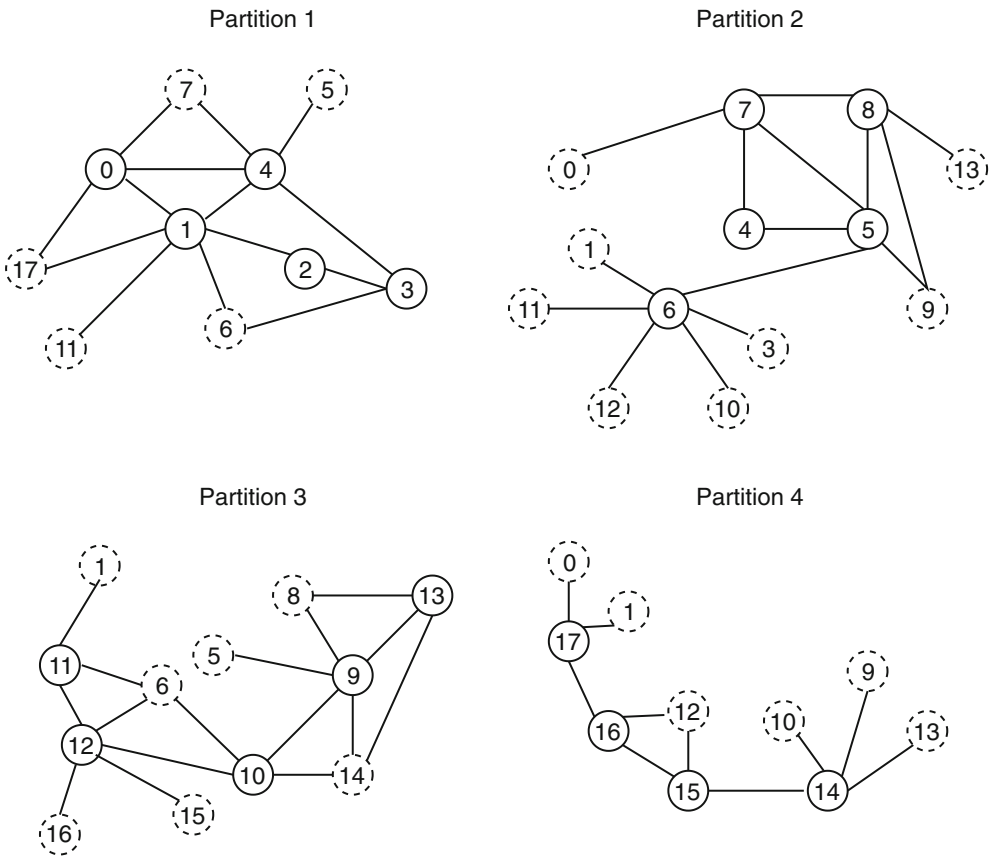

FIGURE 3 Partitioning. Dashed circles represent ghost vertices or neighboring vertices belonging to another partition

While partitioning, each node maintains its border vertices and ghost vertices in a hash map called *ghostList*. Each processor uses the ghost list and marks doubtful vertices that are also border vertices in the ghost list for potential migration to the other processors. In a multi-node distributed memory algorithm, the destination node to which a doubtful vertex has to be moved should be determined. We use a simple heuristic by which the doubtful vertex is moved to a remote processor to which it has the maximum number of cut edges. The non-border doubtful vertices will be treated as isolated communities. Thus, at the end of the migration, each processor will have a new set of vertices. Due to the movement of some border vertices to the other processors, the border list is updated. The algorithm is explained in Algorithm 2. In the algorithm, $G_{new}(V_{new}, E_{new})$ represents the coarse graph containing the communities.

The migration of the doubtful vertices for the graph in Figure 2 is illustrated in Figure 4.

Algorithm 2. Finding ghost vertices and breaking the communities after independent computations in process *i*

Input: $G(V, E)$, $G_{new}(V_{new}, E_{new})$, Community List *Comm* of G, Ghost List *ghost* of G, number of unique communities, *numComm* at process *i*; *pos*=1

```
    /* Checking each Community                                                    */
1   forall c ∈ V_new do
    /* Checking each vertex in a community                                        */
2   for v ∈ c do
3   if v ∈ border[i] and v ∈ doubtful[i] then
    /* vertices marked for movement                                               */
4   isMovement[v] = true
    /* move the vertex to the partition with maximum ghost edges                  */
5   p = remote processor of the border vertex v with maximum ghost edges from ghost list of i.
    move v to p
6   end
7   else if v ∉ border[i] and v ∈ doubtful[i] then
    /* isolate the vertex from the community, so the communityID will change     */
8   commID[v] = numCom + pos
    pos++
9   end
10  end
11  end
```

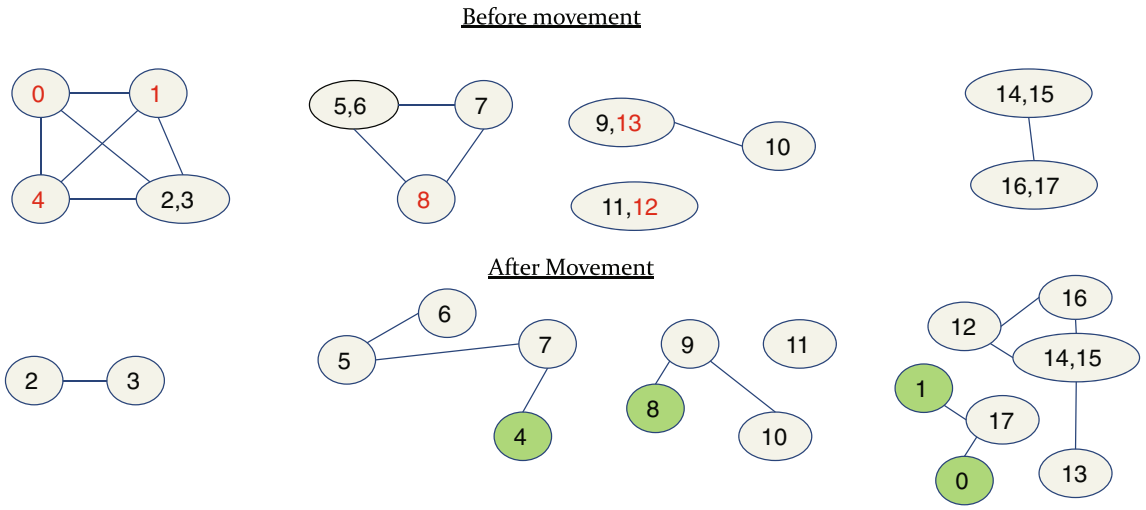

FIGURE 4 Illustration of migration of doubtful vertices. In the figure, the vertices labeled with red color are the doubtful vertices identified for movement. The incoming doubtful vertices to each processor after movement are marked green. Note that some communities change after the movement of doubtful vertices. For example, vertices 2 and 3, that were in the same community before movement are in separate communities after the movement of border doubtful vertices

5.4 | Merging the components from different partitions

There are two ways to merge the coarse graphs from the partitions.

1. One way merging: We transfer coarsened graphs from all the partitions to one partition and merge them using *ghostList* information.
2. Hierarchical merging: We take processors in a pair and then merge them. We continue this procedure until the graph is coarse enough to be accommodated in a single node.

The problem with one way merging is if the graphs in all the partitions are not coarse enough then it is not possible to accommodate the merged components from all the processors in a single node. So, we use a merging strategy proposed by Panja and Vadhiyar[24] for merging components from each processor. According to the strategy, components from each processor are divided into segments and these segments will be exchanged between the processors. This ensures that the components considered for merging in a node can always be accommodated in a node.

As one component may have connection with more than one segments, components are exchanged multiple times. Once the exchange is done, each process has its present component and received component from other processors. For exchange of segments between the processors, ring based communication is used. In a ring based communication, each processor P_i receives segment from its right neighbor and sends the segment to the left neighbor. So, P receives segment from $P_{(i+1)modP}$ and sends its segment to $P_{(i-1)modP}$. Once the segment exchange is done, each processor performs Louvain community detection on its component. This procedure continues until the components of a group can be accommodated in a single node. Group size is fixed as 4 based on experiments. Once all the components from the processors in a group can be accommodated in a group, the components are sent to the *group leader* which performs independent Louvain algorithm on the merged components. This merging strategy is inspired from Rabenseifner's algorithm[25] for reduce and allreduce.

In hierarchical merging, the processors are arranged in groups and the communities in a group are merged in one of the processors of the group called the *group leader*. The group leaders then participate in the next round of independent computations, doubtful vertex migration and merging. This process continues until the final set of communities is formed in a single processor.

The merging of the components for the graph in Figure 2 is illustrated in Figure 5, where the hierarchical merging is shown bottom-up.

5.5 | Putting it all together

All the steps of the multi-node algorithm are explained in Algorithm 3. After Louvain community detection, each node finds *border and doubtful vertices* (as mentioned in the line 6). Once doubtful border vertices are found, they are marked for movement. These marked vertices will be moved to

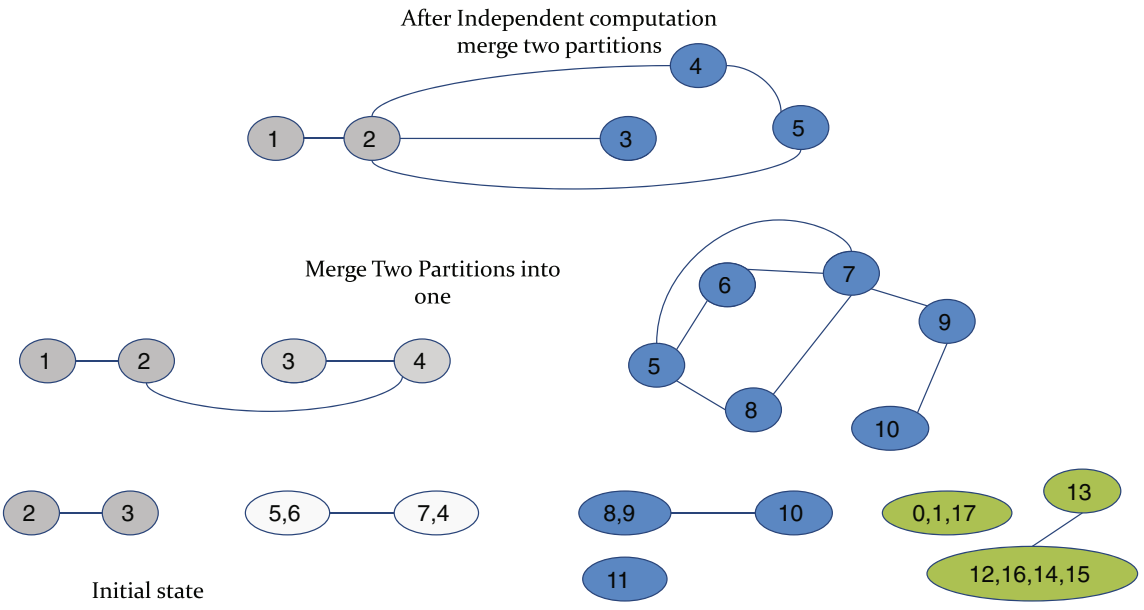

FIGURE 5 Illustration of hierarchical merging of components. The merging starts from the bottom row involving four processors shown by different colors. The bottom row corresponds to the initial communities formed after the migration of doubtful vertices. The different colored components at the bottom row in the merging figure denote the different processor components. The middle row describes the merging of components of processors (0, 1) and (2, 3) to two processors, 0 and 2, respectively. The topmost row indicates the final merging of components from processors (0, 2)

Algorithm 3. The multi-node multi-device Louvain community detection algorithm

Input: GraphG=(V,E), threshold τ ;

Graph at node i ⟵ith Partition of graph (G,E) ;
repeat
  repeat
    $Comm_i$ ⟵ $independentLouvain_i$ of Graph G_i (V_i, E_i) ;
    $NewComm_i$ ⟵ $FindDoubtful_i$ (G_{new} (V_{new}, E_{new})) ;
    $mark_i$ ← $findborderanddoubtful_i$ (G_i) ;
    $Modularity_{old}$ ⟵ find modularity in the coarsened graph ;
    for $v \in mark_i$ **do**
      $target_p$ ← processor with maximum ghost edges with v ;
      move v to $target_p$;
    end
    $Modularity_{new}$ ⟵ find modularity in the new graph ;
  until *(*$Modularity_{new}$ - $Modularity_{old} < \tau$ *)*;
  Merge *newComm* formed in different partitions ;
until *the current coarse graph can be accommodated in single node*;
Move all communities to single node and perform independent Louvain community detection ;

TABLE 1 Graph specifications

Graph	|V|	|E|	Approx. diam.	Avg. deg.	Max. deg.
Twitter	21.00M	265.00M	NA	NA	NA
uk-2002	18.50M	523.00M	29	28.27	194,955
rMat24	16.80M	536.00M	9	31.90	3582
eu-2015	11.20M	759.00M	8	67.42	398,609
gsh	30.80M	1.16B	9	37.73	2.18M
uk-2005	39.40M	1.84B	20	46.69	1.77M
Synthetic	49.79M	3.2B	–	–	–
Orkut	3.00M	110.00M	18	70.00	27,000

Abbreviations: B, billion; M, million.

the processor with maximum cut edges. If both the border doubtful and ghost vertices are marked for movement, there will be only one movement to the lower index processor. After the movement, the change in modularity is found. On convergence of the modularity, the communities are merged. These steps will be repeated until the coarse graph is small enough to be accommodated in a single node.

6 | EXPERIMENTS AND RESULTS

The experiments were performed on our Institute's CrayXC40 supercomputing system where each node is equipped with one Intel Xeon Ivybridge E5-2695 v2 processor and one Nvidia Tesla K40 GPU accelerator card. The CPU processor has 12 cores running at 2.4 GHz with 64 GB main memory. The accelerator card has 2880 cores with 12GB device memory.

The graphs used in our experiments are shown in Table 1. We primarily consider large graphs that cannot fit within the GPU memory. The space complexity of the state-of-art GPU implementation[3] is $\mathcal{O}$(no of nodes in the graph + no of edges in the graph + maximum degree of the graph). The graphs shown in the table were carefully chosen such that their space complexities by the above formula exceed the GPU memory and hence cannot

be executed by the state-of-art GPU implementation. Some of the graphs cannot be accommodated even on multiple GPUs. For these graphs, we use HyDetect, our single-node hybrid algorithm explained in Section 4, for two or more nodes. Some graphs cannot be accommodated in a single-node CPU memory.

The graphs were obtained from the University of Florida Sparse Matrix Collection,[26] the Laboratory for Web Algorithmics,[27,28] and the Koblenz Network Collection.[29] 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[30] was used to generate the rMat24 graph. All the results shown are obtained using averages of three runs.

6.1 | Quality analysis

For the quality analysis, we have used modularity as the metric for our analysis.

6.1.1 | Modularity comparison with single-node Louvain algorithm

Figure 6 shows the comparison of the modularities of our multi-node algorithm with the modularities of the single-node multi-core Louvain algorithm or single-node GPU-only algorithm. If the graph is small enough to be accommodated in a single-node GPU, we compare our algorithm with the single-node Louvain GPU. For large graphs, we compare with CPU multi-core Louvain algorithm.

From the result, it is evident that our algorithm gives comparable modularity with the state-of-the-art single-node Louvain algorithm.[2,3] For *Uk-2005* and *Uk-2002*, we obtain slightly degraded modularities (almost 2%) compared to the single-node algorithm. For the other graphs, we obtain equal or higher modularities compared to the single-node result.

6.1.2 | Sensitivity of modularity to thresholds for defining doubtful vertices

We have experimented with different thresholds for the graphs *Orkut* and *rMat24* to determine the thresholds for defining *doubtful* vertices in each node. In Figure 7, we show the modularity results with 0.2, 0.4, 0.6, and 0.8 thresholds for 2, 4, 8, and 16 nodes. From these results, we can see our algorithm with 0.4 threshold gives the best modularity result for up to four nodes. For higher number of nodes, 0.6 and 0.8 thresholds give better modularities compared to 0.2 and 0.4 thresholds. This is because as we use more number of nodes, the number of border and ghost vertices generated in each node increases, resulting in more number of defective communities formed in the partitioned graph. The threshold captures the degree of uncertainty of a vertex in a community. So, the probability of a vertex being doubtful in the formed community is greater for higher number of nodes. Higher values of thresholds will result in marking the properly placed vertices of a community as *doubtful* and isolating them from their communities resulting in poor quality community formation. Considering these aspects, we have used 0.5 threshold value for the multi-node experiments for up to four nodes and 0.7 threshold value for eight and more number of nodes.

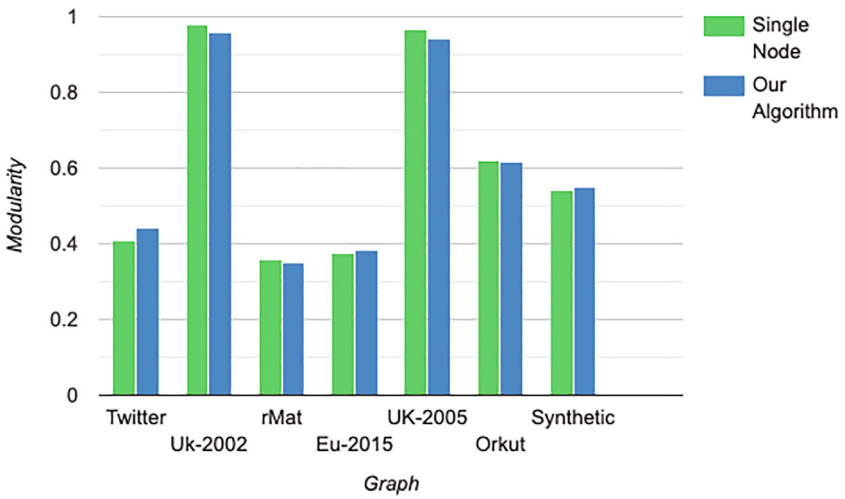

FIGURE 6 Modularity comparison

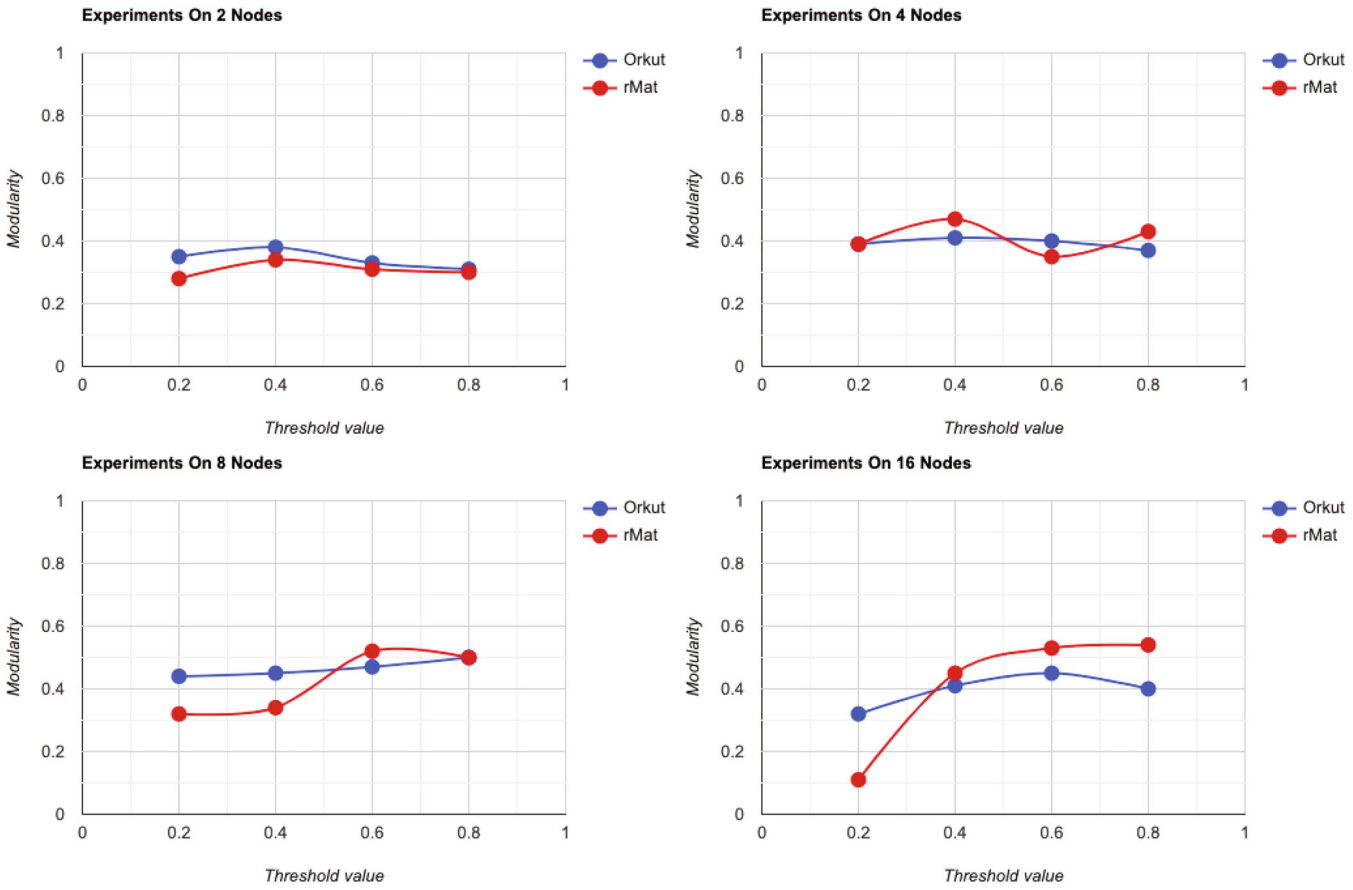

FIGURE 7 Modularity with different thresholds

6.2 | Scalability analysis

Figure 8 shows the scalability of our multi-node multi-GPU algorithm with increasing number of nodes, where each node consists of a single K40 GPU. For this experiment, we show results for up to 24 nodes with 288 CPU cores and 24 GPUs. We find that overall, our algorithm is scalable with increasing number of nodes giving decreasing execution times. For the largest size graphs, namely, *uk-2005* and *Synthetic*, the performance shows large improvements with increasing number of nodes, with up to 7–8× performance improvement over execution on two nodes. This shows that our work is especially applicable for exploration of BigData graphs that are of increasing interest in various domains.

For some of the graphs like *eu-2015*, for large number of nodes, the individual parts formed exhibited less community structure and hence involved more iterations and movement of doubtful vertices to converge to the final communities. Hence, the reduction in times was found to be less beyond certain number of nodes.

6.3 | Speedup over state-of-art single-node algorithm

In our previous work,[6] we demonstrated that our *HyDetect* single-node GPU algorithm gave the best performance over the state-of-art single-node methods. In this section, we compare our multi-node multi-GPU algorithm with our single-node *HyDetect* algorithm. Here, we compare the least execution time of our multi-node multi-GPU algorithm for executions up to 24 nodes with the execution time of single-node *HyDetect* algorithm. Figure 9 shows the results.

From the results, it is evident that our multi-node multi-GPU method works much faster than the single-node version. The least speedup is obtained for *uk-2002* graph which has comparatively smaller size. Apart from the size of the graph, the speedup obtained also depends upon the sparsity of the graph. If the graph is sparse, it takes more time to form communities. For those cases, our multi-node multi-GPU method performs much better irrespective of the size of the graph. For example, *rMat*24 graph generates sparse communities. There is a huge improvement in timing using multi-node multi-GPU method for this graph. In all cases, we obtain 2–47× performance improvement with the multi-node multi-GPU algorithm over the single-node *HyDetect* algorithm.

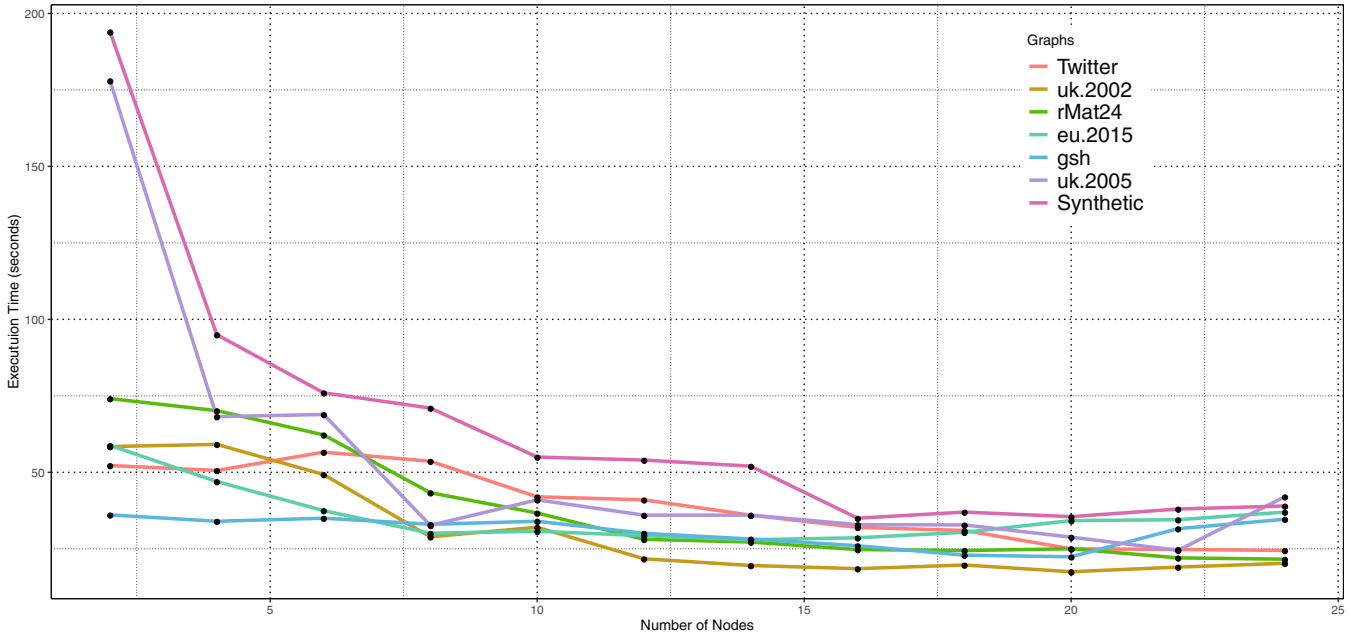

FIGURE 8 Scalability with increasing number of nodes

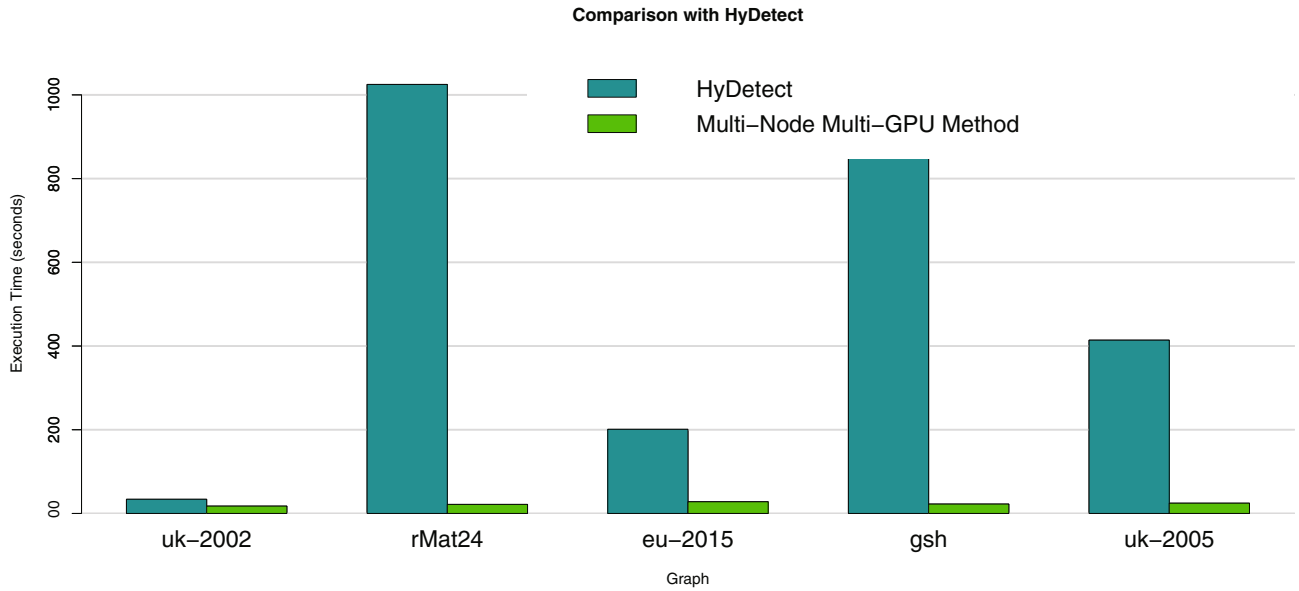

FIGURE 9 Comparison with single-node GPU *HyDetect* algorithm

6.4 | Comparison with state-of-art multi-node algorithm

6.4.1 | Comparison with the work by Ghosh et al.

In this section, we compare our multi-node results with the multi-node Louvain algorithm by Ghosh et al.[5] The work by Ghosh et al. uses only the CPUs. In this work, the graph is partitioned randomly. At the end of each iteration, communication is involved to obtain information about remote communities. The work reduces the 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.

We compared the algorithms on 16 nodes. There are three versions of distributed algorithm in the work by Ghosh et al. We have taken the best version and compared the result with our algorithm for the same number of nodes. For this comparison, we have used the graphs shown in Table 2. These graphs were also used in the results shown by Ghosh et al.

TABLE 2 Specifications of graphs used for comparisons with Ghosh et al.[5]

Graph	$\|V\|$	$\|E\|$	Approx. diam.	Avg. deg.	Max. deg.
Orkut	3.00M	110.00M	18	70.00	27,000
Twitter	21.00M	265.00M	NA	NA	NA
nlpkkt240	27.00M	401.00M	NA	14.33	29
Arabic	22.7M	1.20B	29	55.50	575,662
sk2005	50.26M	3.62B	18	71.49	8,563,816

Abbreviations: B, billion; M, million.

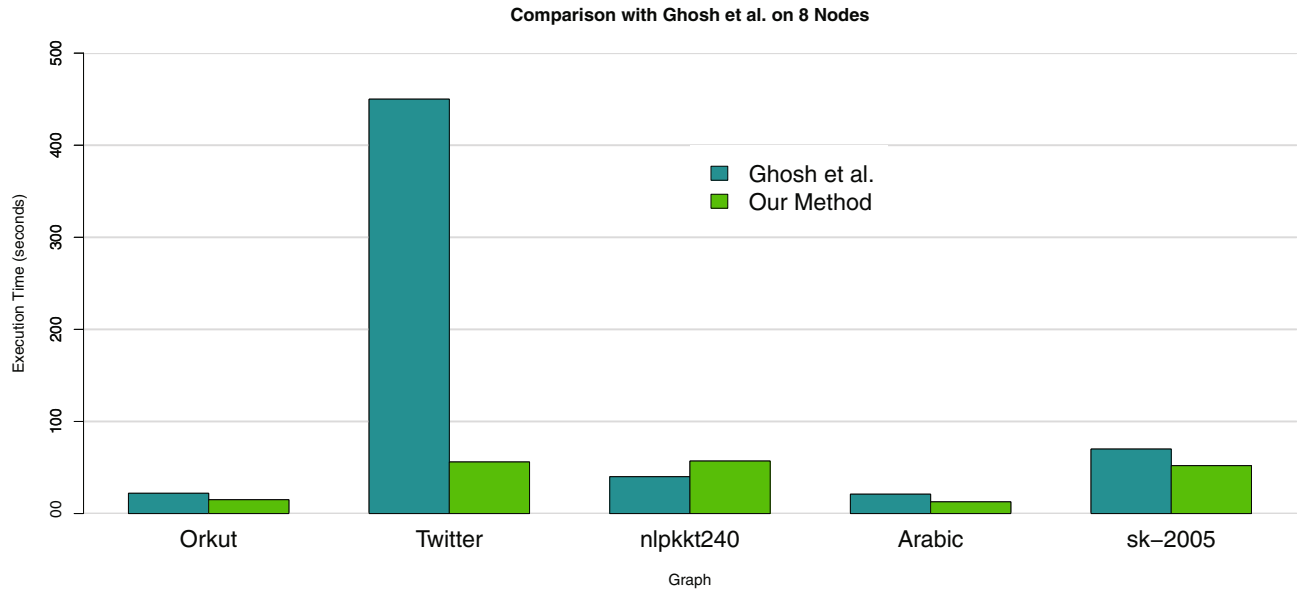

FIGURE 10 Comparison with Ghosh et al.[5]

Figure 10 shows the results. From the results, we find that our method gives smaller execution times or higher performance than the distributed CPU-only Louvain algorithm by Ghosh et al. in most of the cases. If the number of cut edges is less in partitioning, then our method performs significantly better than the distributed CPU only version. For the *nlpkkt*240 graph, we see that our method performs worse than the distributed CPU only version. This graph has very poor edge locality. This leads to sparse graph parts and hence poor community structures from the initial independent Louvain algorithm. There is also large movement of vertices between the processors, where at least 80% of the initial vertices were moved. For the other graphs, our method results in 25%–87% smaller execution times than the method by Ghosh et al.

6.4.2 | Comparison with the work by Cheong et al.

We also compare our results with the work done by Cheong et al.[15] In the work by Cheong et al., each node divides the graph into subgraphs and performs Louvain community detection on each subgraph. The results of the subgraphs are merged using the missing links. The GPU is used only to find neighbor communities and best neighbor community, while the other steps of the Louvain algorithm use multi-core CPU. Figure 11 shows comparison of our algorithm with this algorithm. We can observe from the results that our multi-GPU version has lower computation times for the graphs *Uk2005* and *Orkut*. For the Twitter graph, the method by Cheong et al. has slightly lower execution time compared to our method. However, for this graph, the modularity of their parallel approach shows a degradation of 3.8% compared to the modularity for the single-node execution whereas our algorithm has a modularity increase of 7.01% for the Twitter graph in comparison with the single-node GPU-only execution. In the work by Cheong et al., the average degradation for modularity values in multi-GPU system is around 3%. From Figure 6, we can observe that our multi-node algorithm mostly generates equal or better modularity values for most of the graphs, except in a few cases where the maximum degradation is 2%.

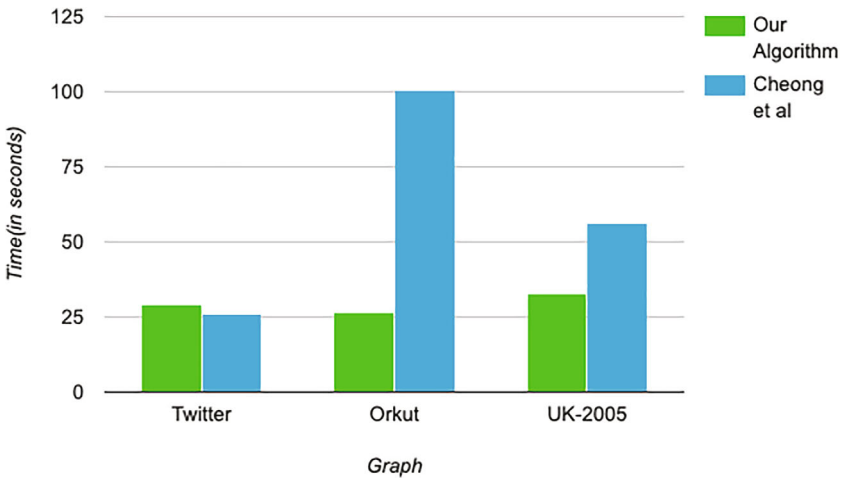

FIGURE 11 Comparison with Cheong et al.[15]

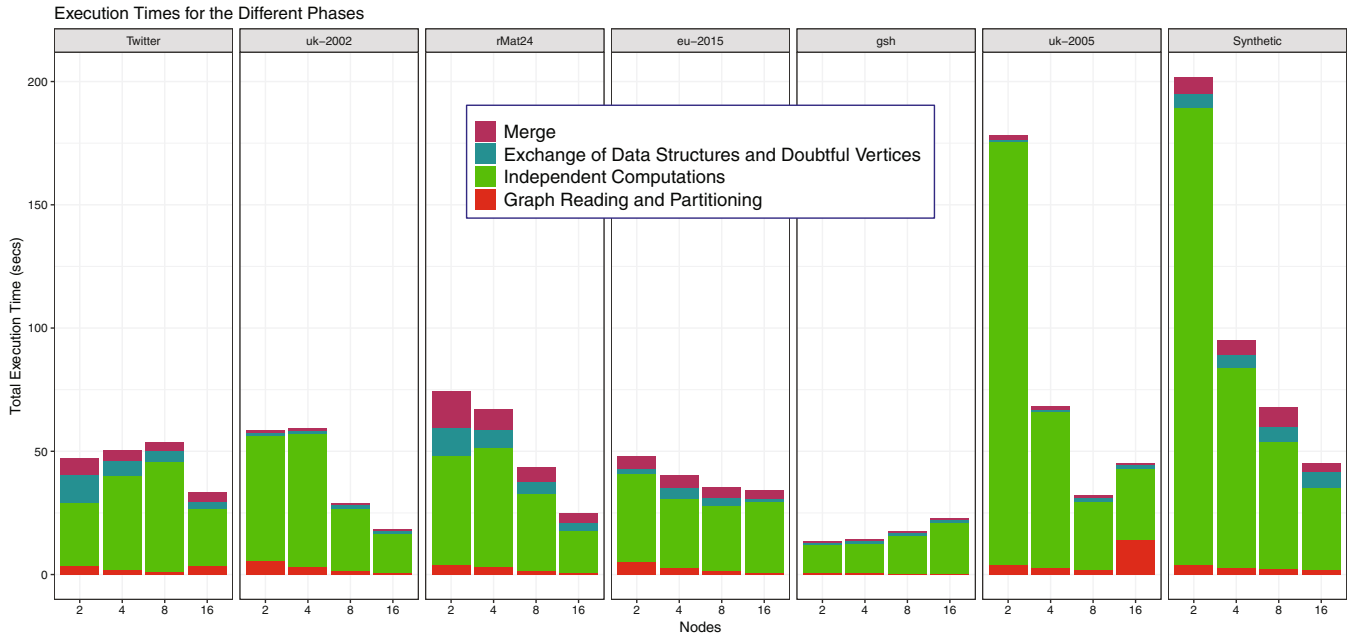

FIGURE 12 Times for various execution phases of our algorithm

6.5 | Execution phases of our algorithm

Figure 12 shows the times for the different execution phases of our multi-node multi-GPU algorithm for different graphs on 2, 4, 8, and 16 node executions.

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 54%–96% of the total time. Note that the independent computations also involve identifying the doubtful vertices and finding the subset of doubtful vertices for migration. The three overheads related to partitioning the graph, communications of the boundary and doubtful vertices between the processors and the hierarchical merging all consume very less time with averages of 5.7%, 7.6%, and 7%, respectively, across all the cases. Thus, our multi-node multi-GPU algorithm spends most of the execution time in useful independent computations on the processors, ensuring large computation to communication ratios and providing good scalability for many graphs.

7 | CONCLUSIONS AND FUTURE WORK

In this work, we have developed a multi-node multi-GPU Louvain community detection algorithm, simultaneously harnessing the CPU and GPU cores of the devices. The algorithm partitions a given graph across multiple nodes and devices in the nodes and performs independent computations

on the parts on the devices. The independently formed communities in the devices are refined by identification of doubtful vertices and migrating them to the other processors. The communities are merged using a hierarchical merging algorithm that ensures at any point the merged component can be accommodated within a processor. Our experiments show that our algorithm is highly scalable with increasing number of devices, provides large-scale performance for BigData graphs and gives up to 87% performance improvement over state-of-art multi node algorithm. Our future work includes development of a generic framework and API for multi-node multi-device executions that can support multiple graph applications.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in University of Florida Sparse Matrix Collection, the Laboratory for Web Algorithmics and the Koblenz Network Collection at the URLs mentioned in References 26-29.

ENDNOTE

*We use the terms nodes and processors interchangeably.

ORCID

Sathish Vadhiyar https://orcid.org/0000-0002-5476-8328

REFERENCES

1. Blondel V, Guillaume JL, Lambiotte R, Lefebvre E. Fast unfolding of communities in large networks. *J Stat Mech Theory Exp*. 2008;2008(10):P10008.
2. Lu H, Halappanavar M, Kalyanaraman A. Parallel heuristics for scalable community detection. *Parallel Comput*. 2015;47:19-37.
3. Naim M, Manne F, Halappanavar M, Tumeo A. Community detection on the GPU. Proceedings of the IEEE International Parallel and Distributed Processing Symposium; 2017:625-634; IPDPS.
4. Que X, Checconi F, Petrini F, Gunnels J. Scalable community detection with the Louvain algorithm. Proceedings of the IEEE International Parallel and Distributed Processing Symposium; 2015:28-37; IPDPS.
5. Ghosh S, Halappanavar M, Tumeo A, et al. Distributed Louvain algorithm for graph community detection. Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium; 2018:885-895; IEEE, IPDPS.
6. Bhowmik A, Vadhiyar S. HyDetect: a hybrid CPU-GPU algorithm for community detection. Proceedings of the 26th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2019; December 17-20, 2019:2-11; Hyderabad, India.
7. Newman M, Girvan M. Finding and evaluating community structure in networks. *Phys Rev E*. 2004;69:026113.
8. Bae S, Howe B. GossipMap: a distributed community detection algorithm for billion-edge directed graphs. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015; November 15-20, 2015:27:1-27:12; Austin, TX.
9. Rosvall M, Axelsson D, Bergstrom CT. The map equation. *Eur Phys J Special Top*. 2009;178(1):13-23.
10. Zeng J, Yu H. A distributed infomap algorithm for scalable and high-quality community detection; ICPP; 2018.
11. Moon S, Lee J, Kang M, Choy M, Lee J. Parallel community detection on large graphs with MapReduce and GraphChi. *Data Knowl Eng*. 2016;104:17-31.
12. Palsetia D, Hendrix W, Lee S, Agrawal A, Liao W, Choudhary A. Parallel community detection algorithm using a data partitioning strategy with pairwise subdomain duplication. In: Kunkel JM, Balaji P, Dongarra JJ, eds. *Proceedings of the 31st International Conference, ISC. 9697*. Lecture Notes in Computer Science. Springer; 2016:98-115.
13. Fazlali M, Morad E, Malazi H. Adaptive parallel Louvain community Detectio on a multicore platform. *Microprocess Microsyst*. 2017;54:26-34.
14. Staudt C, Meyerhenke H. Engineering parallel algorithms for community detection in massive networks. *IEEE Trans Parallel Distribut Syst*. 2016;27(1):171-184.
15. Cheong C, Huynh H, Lo D, Goh R. Hierarchical parallel algorithm for modularity-based community detection using GPUs. Euro-Par parallel processing; 2013.
16. Wickramaarachchi C, Frincu M, Small P, Prasanna VK. Fast parallel algorithm for unfolding of communities in large graphs. Proceedings of the 2014 IEEE High Performance Extreme Computing Conference (HPEC); 2014.
17. Souravlas S, Sifaleras A, Katsavounis S. Hybrid CPU-GPU community detection in weighted networks. *IEEE Access*. 2020;8:57527-57551.
18. Souravlas S, Sifaleras A, Katsavounis S. A parallel algorithm for community detection in social networks, based on path analysis and threaded binary trees. *IEEE Access*. 2019;7:20499-20519.
19. Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J Sci Comput*. 1998;20(1):359-392.
20. Karypis G, Kumar V. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. Technical Report TR-97-061, University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN; 1998.
21. Karypis G, Kumar V. Parallel multilevel K-way partitioning scheme for irregular graphs. Proceedings of the 1996 ACM/IEEE Conference on Supercomputing of Supercomputing '96; 1996.
22. Parau P, Lemnaru C, Potolea R. Assessing vertex relevance based on community detection. Proceedings of the International Conference on Knowledge Discovery and Information Retrieval, KDIR 2015; 2015.
23. Zhu X, Chen W, Zheng W, Ma X. Gemini: a computation-centric distributed graph processing system. Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation; 2016.
24. Panja R, Vadhiyar S. MND-MST: a multi-node multi-device parallel Boruvka's MST algorithm. Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018; August 13-16, 2018; Eugene, OR.
25. Thakur R, Gropp W. Improving the performance of collective operations in MPICH. In: JJ Dongarra, D Laforenza, S Orlando (ed). *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer; 2003.
26. Davis TA, Hu Y. The University of Florida sparse matrix collection. *ACM Trans Math Softw (TOMS)*. 2011;38(1):1.
27. Boldi P, Vigna S. The WebGraph framework I: compression techniques. Proceedings of the 13th International World Wide Web Conference (WWW 2004); 2004:595-601.

28. Boldi P, Rosa M, Santini M, Vigna S. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. Proceedings of the 20th International Conference on World Wide Web; 2011:587-596.
29. Kunegis J. Konect: the Koblenz network collection. Proceedings of the 22nd International Conference on World Wide Web; 2013:1343-1350.
30. Bader DA, Madduri K. Gtgraph: a synthetic graph generator suite; 2006. http://www.cse.psu.edu/%7Ekxm85/software/GTgraph/

How to cite this article: Bhowmick A, Vadhiyar S, PV V. Scalable multi-node multi-GPU Louvain community detection algorithm for heterogeneous architectures. *Concurrency Computat Pract Exper*. 2022;34(17):e6987. doi: 10.1002/cpe.6987