Journal of Electrical and Computer Engineering Innovations
(JECEI)
Journal homepage: http://www.jecei.sru.ac.ir

Research paper

Parallel Louvain Community Detection Algorithm Based on Dynamic Thread Assignment on Graphic Processing Unit

***M. Mohammadi*[1], *M. Fazlali*[2,*], *M. Hosseinzadeh*[3]**

[1]*Department of Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran.*

[2]*Department of Computer and Data Sciences, Faculty of Mathematical Sciences, Shahid Beheshti University, Tehran, Iran.*

[3]*Mental Health Research Center, Psychosocial Health Research Institute, Iran University of Medical Sciences, Tehran, Iran.*

Article Info

Article History:
Received 06 February 2021
Reviewed 28 March 2021
Revised 12 April 2021
Accepted 17 June 2021

Keywords:
Louvain Algorithm
Community Detection
Modularity
Hardware Thread
CUDA

*Corresponding Author's Email Address:
fazlali@sbu.ac.ir

Abstract

Background and Objectives: Louvain is a time-consuming community detection algorithm especially in large-scale networks. Using Graphic Processing Unit (GPU) in order to calculate modularity sigma, which is a major processing section in Louvain algorithm, can reduce algorithm execution time and make it practical for large-scale networks.

Methods: The proposed algorithm Dynamic CUDA Louvain Method (DCLM) blocks hardware threads dynamically on cores inside GPU. By considering the properties of GPU, this algorithm allocates the maximal number of processing cores to each Stream Multi-Processor (SM) as number of threads in a block. If the number of nodes in the graph is smaller than all physical cores on GPU, number of threads per block Is equal to the ratio number of graph nodes over the number of SMs.

Results: The implementation results demonstrated that the proposed algorithm is able to decrease the run time by 15% in comparison with the best past method in the large-scale graph.

Conclusion: We have introduced DCLM algorithm based on GPU that accelerates Louvain community detection algorithm. Dynamic allocation of threads to each block has a significant effect on the reduction of algorithm execution time. However, incrementing the number of threads per block alone does not result to acceleration the speed of calculations.

Introduction

Community detection is one of topics in graph decomposition and network analysis. It has a significance role in many research fields such as social sciences, biology, physics and medical fields. Networks (or Graphs) are usually classified such that edges inside the group have a higher density than edges in between the groups. These important characteristics is called the network structure and the detection of this groups or clusters is called community detection [1]. Each community consists of a number of nodes in graphs that communicate a lot with each other. Nowadays community detection is one of the most important fields in order to comprehend topology and functions in networks [2]. Clustering of nodes in a graph has been a popular method to detect communities in a graph.

Although researchers have implemented parallel algorithms for community detection, it remains time consuming and computationally intensive in large-scale networks. This is because communities have different sizes and the number of communities is usually not defined [3]. Bigger modularity parameter [4], [5]

indicates there exists a larger number of edges inside communities in comparison to edges between communities. It is a parameter and function of a network that evaluates the quality of the classification of graph to clusters.

Parallelizing scalable community detection in large scale graphs with high quality modularity has always been a challenging topic and maximizing modularity is an NP-Complete problem [6].

A number of algorithms have been proposed for untangle community detection in large scale networks [7], [8]. Most of these algorithms employ maximizing the modularity as suitable criteria for community detection. One of this modularity based algorithms is a hierarchical method called Louvain method that Blondel et al. proposed in [9]. This algorithm consists of a number of stages.

At first each node of the graph is considered as a community in each phase. Up until the time communities are stable and the maximum modularity is achieved, the nodes are separately transferred to neighboring communities. A number of papers have been presented that parallelize Louvain method [10], [11], [12], [13], [14], [15], [16].

A Graphic Processing Unit (GPU), having wide computational parallel capabilities, is an appropriate select for speeding-up community detection in large-scale graphs. Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model for Nvidia GPUs. It offers three levels of programming abstractions: thread, thread block, and grid. A warp consists of 32 threads ran together as a working unit in a SIMT manner [17], [18].

In this research, we propose Dynamic CUDA Louvain Method (DCLM) in which the number of threads per block is dynamically calculated and allocated in Louvain algorithm. By considering the properties of GPU, this algorithm allocates the maximal number of processing cores to each Stream Multi-Processor (SM) as number of threads per block.

However, if the number of nodes in the graph is smaller than all physical cores on GPU, it allocates the ratio number of graph nodes over the number of SMs as number of threads per block. Considering the fact that in this algorithm the number of threads per block depends on the number of nodes in the networks, shared memory per block is adaptively defined. We summarize the main contributions in this paper as follows:

- Presenting DCLM parallel algorithm based on GPU.
- Calculating the optimal number of threads and allocating them dynamically to each block.
- DCLM algorithm, calculates the number of required SMs dynamically to calculate the modularity on GPU. This results in using less number of threads in comparison to previous methods and reducing paralleling overhead.
- The proposed DCLM needs less memory usage than the previous algorithm.

Implementation results indicates effective paralleling of Louvain can be achieved by dynamic allocation of the threads per block to reduce the executing time of the algorithm.

Background

Here at first we explain Modularity parameter which is important quality function in Louvain algorithm. Afterwards Louvain algorithm is clarified. Then, previous methods on paralleling of Louvain are reviewed.

A. Modularity

In order to evaluate the community structure in a network, modularity is used as a quality function and here it is shown as the letter *Q.* Based on the modularity criteria, community detection in a graph is done successfully when a great number of edges exist inside the communities and smaller number of edges exist in between communities.

In strategies based on modularity, the goal is to find a partition in the graph that has the maximum value of *Q.* Modularity has a value between -1 and 1. The closer *Q* gets to 1 the better the communities are divided but it never reaches 1. In practice a modularity of $Q > 0.3$ shows a suitable structure.

Consider Graph $G= (V, E)$ is an undirected graph in which $n=|V|$ shows the number of nodes and $m= |E|$ shows the number of edges in the graph. Graph *G* is described by matrix $A = (A_{ij})$. Whenever node *i* is linked to node *j* their corresponding index in the matrix equals one, otherwise it equals zero. Also K_i is the degree of node *i.* Modularity of the partitioning $C= \{C_1, C_2, \ldots C_L\}$ of graph *G* is calculated by (1) [19].

$$Q(C) = \frac{1}{2m} \sum_{i,j \in v} \left(A_{ij} - \frac{k_i k_j}{2m} \right) \delta_{i,j} \quad (1)$$

If node *i* and node *j* exist in the same community the value of $\delta_{i,j}$ equals one, otherwise it equals zero. Discovering the maximum value of *Q* is a NP-hard problem. Therefore, a heuristic solution that concurrently guarantees scalability and reasonable calculation costs is needed.

In community detection algorithms usually it is considered the edges are undirected and it is presumed that the graph is undirected.

This results in a significant improvement in partitioning the system. Until now different methods have been presented for maximizing the value of modularity [20]. Guo et al. developed a family of generalized modularity measure, f-Modularity, which includes the original modularity as a special case [21].

One of the most famous algorithms is a greedy algorithm that was presented by Newman and Girvan [4]. This algorithm starts by putting each node in a separate community. At the start of this algorithm all nodes are communities connected by edges. By adding the edges one by one the communities that exists on the two sides of the edges are merged if it results in an increase in modularity. If by adding, an edge a merge does not happen between the communities, that edge is an inner edge of the community. Therefore, the modularity is not changed. The number of the divisions found in the process equals the number of nodes. Each of this divisions have a specific modularity. After adding the edges, the division that has the largest modularity is chosen as the output.

B. Louvain Community Detection Method

Between community detection methods based on modularity, Blondel et al. [9] presented a heuristic method called Louvain that has a lower time complexity. This algorithm includes two phases and they are repeated iteratively. In starting, each node in graph is assumed as a community.

Therefore, a high number of communities exists in the first partitioning. Then for every node *i* all of its *j* neighbors are considered and modularity is computed. If modularity increases node *i* is added to its neighbor *j*, otherwise it remains as an initial community.

This process is reiterated consecutively for the nodes in graph until no extra increase on modularity is achieved.

In the seconds phase new communities are created from the communities in the first phase. In this phase each community is considered as a super node and the inner edge's weights of the super node is demonstrated as a loop. In addition, the weight of the communication edge between each super node to its neighboring super node (outer edge) equals to the weights of the node's edges to its neighboring super node. After the completion of the second phase, the algorithm may be repeated again.

The effectiveness of the algorithm arises from the fact that the benefit of the modularity ΔQ is obtained from the transition of node *i* into community *C* which is demonstrated in (2):

$$\Delta Q = \left[\frac{\sum_{in} + K_{i,in}}{2_m} - \left(\frac{\sum_{tot} + K_i}{2_m}\right)^2\right] - \left[\frac{\sum_{in}}{2_m} - \left(\frac{\sum_{tot}}{2_m}\right)^2 - \left(\frac{K_i}{2_m}\right)^2\right] \quad (2)$$

In (2) $\sum_{in}$ is the total of the weight for the internal edges of community *C* also $\sum_{tot}$ is the total of the weight for all the edges that have one end to community *C*. K_i is the total of the weights for the edges that enter node *i* and $K_{i,in}$ is the total of the weight of the edges which exist from node *i* to inner nodes of the community *C*. *m* is the sum of the weights for all edges inside the graph. The next phase consists of creating a new graph that nodes are the created communities in the previous phase. In order to perform so the weight of the edges in between the new nodes are created from the weight of the edges between the two communities. The edges between nodes from the same community result to loops in that community in the new graph. Afterwards the finishing the second phase, again the first phase is applied to the weighted graph and this operation is to be iterated.

Related Work on the Paralleling of Louvain Community Detection Method

Decomposition and analysis of large-scale graphs by Louvain algorithm needs a powerful machine. This is commonly costly and is not affordable. hence, researchers have tried to speed up the algorithm exploiting parallel platforms.

Meyerhenke and Stuadt [11] have proposed an algorithm named Parallel Louvain Method (PLM) based on OpenMp using label propagation in which the transition of nodes in the Louvain is performed in parallel. In [22] researchers have presented a framework which benefits from the overlapping of shared memory called Ensemble Preprocessing (EPP) for community detection. In this framework Parallel Label Propagation (PLP), which was presented by Raghavan et al. [23], used as the basis algorithm. Parallel Louvain Method with Refinement (PLMR) [22] adds a reformation to each level of Louvain to increase the modularity but it slightly increased the algorithm execution time.

To reduce the time of the first round of the Louvain method, Carnivali et al. [24], proposed a Coarse-Grained Vertex Clustering (CoVeC) method. CoVeC pre-process the original graph in order to forward a graph of reduced size to the Louvain method. According to their evaluation, CoVeC outperforms the Louvain method and its variations, attaining a mean execution time reduction of 47% and a mean modularity reduction of only 0.4%.

In [25] Que et al. proposed an algorithm for paralleling of Louvain that improves convergence, modularity and quality of the detected communities. The main aspect of this algorithm is a general strategy for coordination of transfer of vertices to neighbor communities using the adaptive threshold that is executed exponentially in each iteration of the inner loop.

In [26] Zeng and Yu proposed a graph clustering framework for Louvain algorithms based on an asynchronous approach. Also in [27], a Gradient Descent

framework of modularity optimization called vector-label propagation algorithm (VLPA), where a node is associated with a vector of continuous community labels instead of one label, have been proposed.

In [28] Sarmento has tried to apply a density optimization of communities found by the label propagation algorithm and has investigated what happens regarding modularity of optimized results. He introduced a metric called Average Density per Community (ADC).

Also a novel model, named Modularized Deep Non Negative Matrix Factorization (MDNMF) for community detection, which preserves both the topology information and the instinct community structure properties of the community, has been proposed in [29].

In our previous research [10] we have utilized a course-grained paralleling method in order to accelerate Louvain algorithm on multicore systems.

This method uses a processing thread in order to compute the modularity for each neighbor node in the graph. Then, we proposed a parallel algorithm at thread level in [14], for accelerating Louvain algorithm on multicore systems.

This method assigns threads adaptively in parallel to compute the modularity of adding eligible neighbor nodes to the community.

The algorithm finds the idle cores and obtains the appropriate number of threads to calculate the sigma in the modularity formula. This results in improving speed-up in comparison to previous methods while the quality of the resulted modularity remains approximately the same.

Cheong et al. presented the GPU-based parallelized Louvain algorithm in [12]. It uses three levels of paralleling in Louvain algorithm and uses Single-GPU and Multiple-GPU platforms.

Evaluating the method in [12] on multiple large web based graphs and popular social networks show that it is able to speed up community detection however it leads to 3% decrement in the quality of the community detection. SOURAVLAS et al. [30] presented an extension of threaded binary tree approach for community detection. They share the computational load between the two units: the CPU takes specific samples of the network communities and organizes them in the form of threaded binary trees. The GPU takes over the heavy load of reading this data and transforming it into a path-matrix. Finally, this matrix is sent back to the CPU for analysis, community detection and overlaps, as well as network information upgrades.

Shoa et al. [31], suggested the attractor technique which uses distance metric instead of similarity. However, the suggested technique is slower than Louvain method. Zhu et al. proposed a new central node indicator and a new modularity function in [32]. Authors firstly suggested a new local centrality indicator (LCI) to extract local important nodes that are well distinguished from their neighbors. Then, they proposed a new local modularity function F2. F2 can overcome certain problems of other modularity functions such as the resolution limit problem. In [33] Guti´errez et al. proposed a modification of Louvain algorithm, which allows to take into account some additional information about affinity among nodes when detecting communities. This additional information is defined by a fuzzy measure, μ, which is aggregated to the weight of the classical relations among nodes when optimizing the modularity of the clusters throughout the algorithm.

In [34] Miasnikof, et al. have described a new set of statistically rooted clustering quality measures that allow formal clustering quality assessments and comparisons of clustering algorithm performances. Their measures are shown to be more robust than the commonly used modularity and conductance. In [35], Jin et al. have proposed a new MRF approach, namely ModMRF, to formalize modularity as the energy function for community detection in undirected static networks. ModMRF has reduced the time complexity to a nearly linear case.

In [36] a distributed scalable community detection method was presented by Konstantinos et al., that is according to the combination of the characteristics of the topology of graph. In [37] Palacio-Nino and Berzal have advocated for the use of the local network structural properties employed by local link prediction methods in hierarchical community detection. Mahabadi and Hosseini [38] presented an online parallel overlapping community detection approach based on a speaker-listener propagation algorithm by proposing a novel parallel algorithm and applying three new metrics. This approach is presented to improve modularity and expand scalability for getting a significantly speedup in low time-consuming and usage memory through an agent-based parallel implementation in a multi-core architecture. But this algorithm is the overlapping method which is different from Louvain which is non-overlapped method.

Naim et al. [39], suggested a GPU-based Louvain algorithm. It works on load balancing by scaling the number of threads assigned to each node, according to node's degree. In the first phase of the algorithm, it partitions the nodes to subsets based on their degrees. Then the algorithm computes and updates the destination community of each vertex by using a various number of threads per node for the set. The defect of this method is high memory usage. Here we have tried to do this by presenting a new parallel Louvain algorithm. Hence, we presented Adaptive CUDA Louvain

Method (ACLM) [40], which employs GPU to accelerate Louvain method. ACLM algorithm computes the modularity sigma from by adding a community with the neighbor nodes in parallel, in a fine-grained way, by CUDA cores on GPU. This can reduce the overhead of paralleling and speed up the computation of modularity.

ACLM assigns threads per block according to the number of SMs and warps needed to calculate modularity. This algorithm, computes the number of needed SMs before assigning hardware threads. This is because processing time of the warps is fast. Therefore, ACLM obtains the number of thread in the block according to the multiplication of warps. This result of executing ACLM on various web-based graphs demonstrates which it can successfully accelerate algorithm execution time up to 77% in comparison to the past paralleling of Louvain method in the large graph benchmarks.

The Proposed Dynamic CUDA Louvain Method Algorithm

In order to accelerate Louvain community detection method, we propose DCLM algorithm, which employs CPU and GPU on the computational heterologous processing platform. It calculates the modularity of graph using cores in GPU. Efficient allocation of threads per block to compute modularity sigma in this algorithm can decrease the algorithm run time.

A. Investigating the Number of Neighbors for Vertices in Each Pass of the DCLM Algorithm

Blocking and assigning the desired number of threads per block leads to an effectual decrement in the run time of the DCLM algorithm to find communities. Increasing the number of threads per block may not increase the speed of the algorithm.

In each phase of the Louvain algorithm, the communities are formed randomly. Before running the algorithm, it is not possible to accurately predict how many processing threads should be assigned to each block to obtain the best result.

I dynamically based on the multiplication of warps and considering the maximum number of CUDA cores. On the NVIDIA Tesla K20Xm graphics card, each SM has 192 CUDA hardware cores, every block has the maximum number of 1024 threads, and the warps have 32 threads.

Therefore, we categorized the number of graph vertices in each pass of the DCLM algorithm according to the number of their neighbors and extracted the graph vertex statistics. Fig. 1 and Fig. 2 show the number of graph vertices that have 0 to 32, 33 to 192, 193 to 1024 and more than 1024 neighbors in the smallest graph and the largest graph, respectively.

As Fig. 1 shows, in CNR-2000 benchmark, the first three passes have a major number of graph nodes, and most nodes have 0 to 32 neighbors. In subsequent passes, although the number of vertices with 0 to 32 neighbors is low, the number of graph vertices is much less than the initial passes. In this benchmark, there are a total number of 349951 nodes in the total number of passes in DCLM algorithm. 85.4% of vertices have up to 32 neighbors, 12.5% have 33 to 192 neighbors, 1.5% have 193 to 1024 neighbors and 0.6% have more than 1024 neighbors.

Fig. 2 shows the number of graph vertices based on the number of neighbors in 8 passes for the large Cage-15 benchmark. In all passes of the algorithm, the vertices have more than 0 to 32 neighbors. In this benchmark, there are 394,303,332 vertices in all passes in DCLM algorithm, of which 82.9% have up to 32 neighbors, 7% have 33 to 192 neighbors, 5.5% have 193 to 1,024 neighbors, and 4.6% have more than 1,024 neighbors.

B. Dynamic CUDA Louvain Method (DCLM)

In DCLM algorithm, the modularity of the network is computed using cores in a fine-grained manner and threads execute the items in the modularity sigma in parallel manner. This algorithm usages the base modularity in (3) [41].

$$Q = \sum_i (e_{ii} - {a_i}^2) \tag{3}$$

In (3) *i* is the indicator of a supernode. Modularity for each supernode equals ${e_{ii}} - {a_i}^2$. ${e_{ii}}$ is the ratio of the number of edges that connect the inner nodes of the community *i* over all the edges of the graph with random distribution.

In other words, it equals the number of internal edges of the supernode *i*. We consider ${e_{ij}}$ to be half of the ratio of edges which link the nodes of community *i* with community *j*. Therefore ${a_i} = \sum_j {e_{ij}}$ equals the ratio of the number of edges which have at least one node in community *i* over all the edges of the graph with random distribution. To be more clear ${a_i}$ equals the number of outer edges of supernode *i*.

The calculation of the resulted modularity of merging two neighbor nodes is done using cores available in SMs on GPU and other calculations are done sequentially on the CPU.

According to the number of cores in the graphics card of the system, which is divided on the SMs and operates in parallel, the maximum number of threads per block is defined as the same number of processing cores of each SM. Although each block can have more threads but each SM can process the block in parallel at the same time as the same number of its processing cores. Therefore, it would be better to use the maximum cores of all SMs at the same time.

If the number of nodes in the graph is more than all

the physical cores of the graphics card of the system, the computation of the items of the modularity sigma is performed concurrently using the cores in SMs. In this situation we propound the maximum number of threads in the block to be equal to the maximum number of cores in each SM. If the number of nodes is less than the number of cores, we allocate the threads to each block in proportion to the number of nodes over the number of SMs.

In DCLM algorithm, all threads are divided to blocks and in each block there exists a shared memory between the threads of the block. Threads in the block can use data inside the shared memory or store their calculation results inside it.

Considering the fact that the number of threads vary adaptively in this algorithm, we define the shared memory adaptively.

This results in an optimal usage of the shared memory. When calculating the modularity, we consider the graph to be compromised of super nodes. We have the total of the weights of the self-loops of any supernode in vector *in* and the total of the weights of all edges of each supernode in vector *total_weighted*. In fact vector *in* equals $\sum_i e_{ii}$ and vector *total_weighted* equals $\sum_i {a_i}^2$.

Algorithms 1 and 2 demonstrate the stages of DCLM algorithm.

In algorithm 1, on line 3, total of weights for all edges at graph is stored into *g.total_weight* variable. On line 6, degree of node *i* is stored in *size* variable. On line 7, total of weights for internal edges of each neighbor node is stored in vector *in*.

On line 8, total of weight of outer edges of each neighbor node is stored in *total_weighted* vector. The number of SMs required is specified in line 11. On lines 12 and 13 the number of threads per block and on line 14 number of blocks per grid are set. The size of shared memory is determined in line 15. After that in the second algorithm, modularity is calculated in parallel using GPU cores.

In algorithm 2, CUDA kernel is called by input parameters and on line 9, items of modularity sigma are computed. Then computing the modularity by threads in block, on line 13 threads are synced and on line 14 total of the amount of computed modularity in blocks is saved in *quality* vector. Afterwards the *quality* vector is sent to CPU.

On line 2 of algorithm 2, the calculated modularity value in an SM is achieved by dividing the total of computed modularity amounts in blocks (*quality* vector) over the total of weights of all edges of the graph (*g.total_weight*). Then modularity of supernode v_i is obtained by adding the computed modularity amounts in SMs. In Continue algorithm 1 (lines 18 onwards), if the modularity increases, node *i* is attached to the *j* community. This action is done frequently and consecutively for all nodes until no further improvements can be made.

Algorithm 1: DCLM G(V, E)

1: **repeat**
2: $N_C \leftarrow G$
3: g.total_weight ← Total of the weights of all the connections in the N_C
4: **for** all nodes in N_C **do**
5: Choose a random v_i
6: size ← Degree (v_i)
7: in [] ← self-loops of each neighbor's j of node i
8: total_weighted [] ← weighted degree of each neighbor's j of node i
9: $N_{SM} \leftarrow number\ of\ SMs\ in\ GPU$ *//NVIDIA Tesla K20Xm architecture has 14 SMs*
10: $K \leftarrow number\ of\ cores\ in\ each\ SM$ *//NVIDIA Tesla K20Xm architecture has 192 cores in each SM*
11: $\text{stream\_number} \leftarrow \min\left\{N_{SM}, \left\lceil\frac{size}{k}\right\rceil\right\}$ *// number of streams (needed SMs)*
12: $s \leftarrow \frac{size}{stream\_number}$
13: $\text{threadsPerBlock} \leftarrow \begin{cases} 32 & 1 \le s \le 32 \\ 64 & 32 < s \le 64 \\ 128 & 64 < s \le 128 \\ 192 & otherwise \end{cases}$
14: $\text{blocksPerGrid} \leftarrow \left\lceil\frac{s}{\text{threadsPerBlock}}\right\rceil$
15: $\text{sharedMemSize} \leftarrow \text{threadsPerBlock}$
16: **while** true **do**
17: mod_{new} ← Compute Modularity ()
18: **if** $mod_{new} <= mod_{current}$ **then**
19: break
20: **end if**
21: $mod_{current} \leftarrow mod_{new}$
22: G ← new input graph
23: **end while**
24: **end for**
25: **until** No improvement of modularity

Algorithm 2: Compute Modularity (Inputs)

Inputs:

blocksPerGrid:	*// Number of blocks per grid.*
threadsPerBlock:	*// Number of threads per block.*
sharedMemSize:	*// Size of shared memory.*
streams:	*// Vector of needed SMs*
total_weighted:	*// Vector of the weighted degree of any $v_j \in \{(v_i, v_j) \in E\}$.*
in:	*// Vector of self loops of any $v_j \in \{(v_i, v_j) \in E\}$.*
g.total_weight:	*// Total of the weights of all the connections in NC.*

quality: *// Vector of Modularity sections.*
size: *// Degree of vi.*
stream_number: *// number of streams (needed SMs)*

Output:
Resulting Modularity

1: Modularity Kernel <<<blocksPerGrid, threadsPerBlock, sharedMemSize, streams[stream_number]>>> (total_weighted, in, g.total_weight, quality, size, stream_number);
2: Modularity _Value_SM ← sum of quality vector / g.total_weight;
// The Modularity_Value_SM variable includes the calculated modularity values in an SM.
3: Resulting Modularity ← Total of the computed modularity amounts in SMs.
4: return Resulting Modularity

In Modularity Kernel :

5. **for** each neighbor j of node i
6: tid = threadIdx + blockIdx*blockDim
// The amounts of the variables "threadIdx", "blockIdx", "blockDim" and "gridDim" come from input parameters "threadsPerBlock" and "blocksPerGrid".
7: extern __shared__ float cache []
8: **while** (tid < size) **do**
9: temp += in[tid] - (total_weighted [tid] × total_weighted [tid]) / g.total_weight
10: tid += blockDim*gridDim
11: **end while**
12: cache[threadIdx] = temp
13: syncthreads ()
// This call assurances that each thread in the block has perfected commands. Prior to __syncthreads() before the hardware will run the next command on every threads.
14: atomicAdd (&quality[stream_number], cache);
// Sum of values in cache vector is stored in quality vector.
15: return quality

Results and Discussion

We compared our proposed algorithm with four of the parallel techniques for Louvain community detection algorithm.

The first one is PLM [22] that computes the modularity in parallel. The second one is APLM [14] in which allocates the appropriate number of execution threads on CPU and calculate modularity of each two neighbor communities adaptively. The third one is NAIM, a previous GPU-based Louvain method presented in [39]. The last algorithm is ACLM [40] that uses the shared memory on GPU, as well as the optimal number of threads on GPU blocks. Table 1 and Table 2 demonstrates the specifications of our implementation platforms that we ran all the algorithms.

We utilized CUDA in order to run DCLM, ACLM, and NAIM algorithms in the GPU and nvcc compiler was used for compiling the codes. Nvidia's latest CUDA enables a diversity of synchronization techniques. The specifications current version of CUDA (10.0) is described in [17].

For compare our results with APLM and PLM algorithms these algorithms were executed on CPU by using gcc compiler version 4.8.2 on Platform 1 and using gcc compiler version 7.4.0 on Platform 2. The algorithms are implemented with C++. Large graphs of the real world include heterogeneous data that leads to the fact that selecting an ideal sample data and popularizing it to be infeasible. We have used four benchmarks that particularities are offered in Table 3. The graphs have various sizes from small size similar to CNR-2000 to scale graphs same as Cage-15.

CNR-2000 benchmark is a small crawl for Italian CNR domain, mainly useful for debugging and testing purposes. EU-2005 benchmark is a small crawl of the .eu zone.

This graph displays a very low locality, perhaps because the crawl was small and the elected domain is dummy howsoever. IN-2004 benchmark is a small crawl for .in area performed for the Nagaoka University of Technology. Cage-15 graph is the DNA electrophoresis, 15 monomers in polymer. A. van Heukelum, Utrecht U.

A. Comparison of Algorithm Execution Times

Here for a fair comparison, we selected those related works that employ paralleling to accelerate the Louvain method. We compare our proposed DCLM algorithm with the parallel Louvain algorithms based on algorithm execution time.

Parallel Louvain Method (PLM) and Adaptive Parallel Louvain Method (APLM) are the fastest parallel Louvain method that use multicore system. The previous GPU implementation of Louvain was proposed in [39] (NAIM). ACLM an Adaptive CUDA Louvain Method algorithm which profits from the graphic processing unit. The execution times of DCLM, ACLM, PLM, APLM, and NAIM algorithms on the Platform 1 are demonstrated in Table 4 and running times on Platform 2 are demonstrated in Table 5.

With attention the results presented in Table 4, run time of the DCLM algorithm by using 2688 processing cores in platform 1 is better than other algorithms. Reduction in the run time of this algorithm in comparison to PLM is 97% in CNR-2000, 92% in EU-2005 and 89% in IN-2004 and 81% in Cage-15. As well as the reduction in run time of DCLM in comparison to APLM

are 40% in CNR-2000, 43% in EU-2005, 30% in IN-2004 and 31% in Cage-15. The results indicate that the paralleling overhead of using GPU in DCLM algorithm is proper and it defeats the implementations of multicore platforms.

However, NAIM algorithm cannot overcome multicore implementation effectively for benchmarks. The main reason behind this is the communication overhead of GPU platform.

The run time of DCLM algorithm in evaluation to NAIM algorithm is better 78% in CNR-2000, 79% in EU-2005, 51% in IN-2004 and 46% in Cage-15. This demonstrates the DCLM algorithm is better than NAIM algorithm.

The reason is adaptively assign of the cores on GPU to threads in our algorithm.

DCLM algorithm running time is reduced by 10% in the CNR-2000 and EU-2005 benchmarks, by 11% in the IN-2004 and by 15% in the Cage-15 compared to the ACLM algorithm.

With attention, the results presented in Table 5, the run time of the GPU-based implementations are better than multicore implementation for all benchmarks. This demonstrates that utilizing GPU to speed up Louvain community detection is suitable manner in modern GPU platforms.

The proposed DCLM algorithm outperform NAIM algorithm for all benchmarks. Although dynamic thread creation in the proposed method has extra overhead, it can have a better load balancing in comparison to the previous (especially NAIM) methods. We could not run the NAIM algorithm on platform 2 for Cage-15 benchmark because of its memory usage in the biggest graph.

Nevertheless, the DCLM algorithm can execute and end the program on the platform. Again it shows that the proposed DCLM algorithm needs less memory usage than the NAIM algorithm in Cage-15 benchmark. Again this demonstrates the effectiveness of dynamically allocating cores in SMs to the threads. The running time of the DCLM algorithm in CNR-2000 12%, in the EU-2005 11%, in the IN-2004 14% and in the Cage-15 15% is less than ACLM algorithm.

B. Comparison of modularity

While DCLM algorithm leads to acceleration of Louvain method, Table 6 and Table 7 demonstrate that the modularity values that are achieved from running this algorithm is comparable and near the same as the results of other algorithms.

Generally, these algorithms are very resisting to changes in modularity and random communities that are achieved have a similar modularity.

Conclusion

In this paper, we have suggested DCLM algorithm based on GPU, which accelerates Louvain method. In DCLM algorithm, by using a heterogeneous CUDA calculation model and benefiting from the processing cores of GPU, calculation of the modularity resulted from merging two neighbor communities is done in parallel. By optimal usage of the shared memory inside blocks, DCLM algorithm allocates the optimal number of processing cores in SM as number of threads per block.

The evaluation results on a number of large scale web based graphs demonstrates that the proposed algorithm reduces the execution time up to 15% in comparison to the best previous parallel implementations of Louvain method. The results also demonstrate that dynamic allocation of the number of threads to each block has a notable result on the reduction of execution time of the algorithm, but incrementing the number of threads per block alone does not result to acceleration the calculations.

Author Contributions

M. Mohammadi carried out the design and implementation of the algorithm and wrote the manuscript. M. Fazlali and M. Hosseinzadeh explicate the outcomes and contributed to edit the manuscript.

Acknowledgment

We thank the editor and all anonymous reviewers.

Conflict of Interest

The author declares that there is no conflict of interests regarding the publication of this manuscript. In addition, the ethical issues, including plagiarism, informed consent, misconduct, data fabrication and/or falsification, double publication and/or submission, and redundancy have been completely observed by the authors.

Abbreviations

CUDA	Compute Unified Device Architecture
SM	Stream Multi-Processor
DCLM	Dynamic CUDA Louvain Method
ACLM	Adaptive CUDA Louvain Method
APLM	Adaptive Parallel Louvain Method
PLM	Parallel Louvain Method
PLMR	Parallel Louvain Method with Refinement
PLP	Parallel Label Propagation
EPP	Ensemble Preprocessing

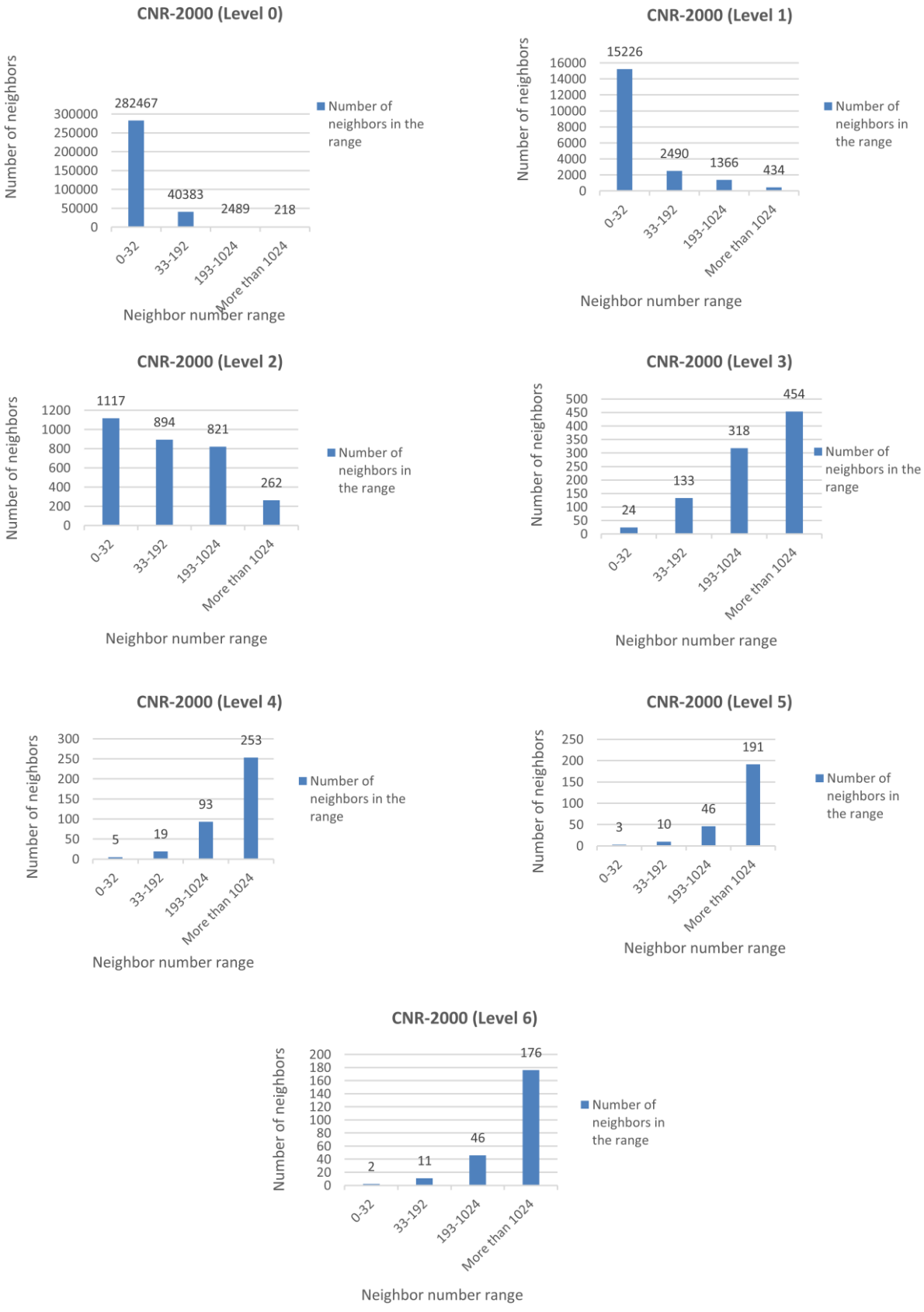

Fig. 1: Classify the number of neighbors of nodes in the CNR-2000 benchmark.

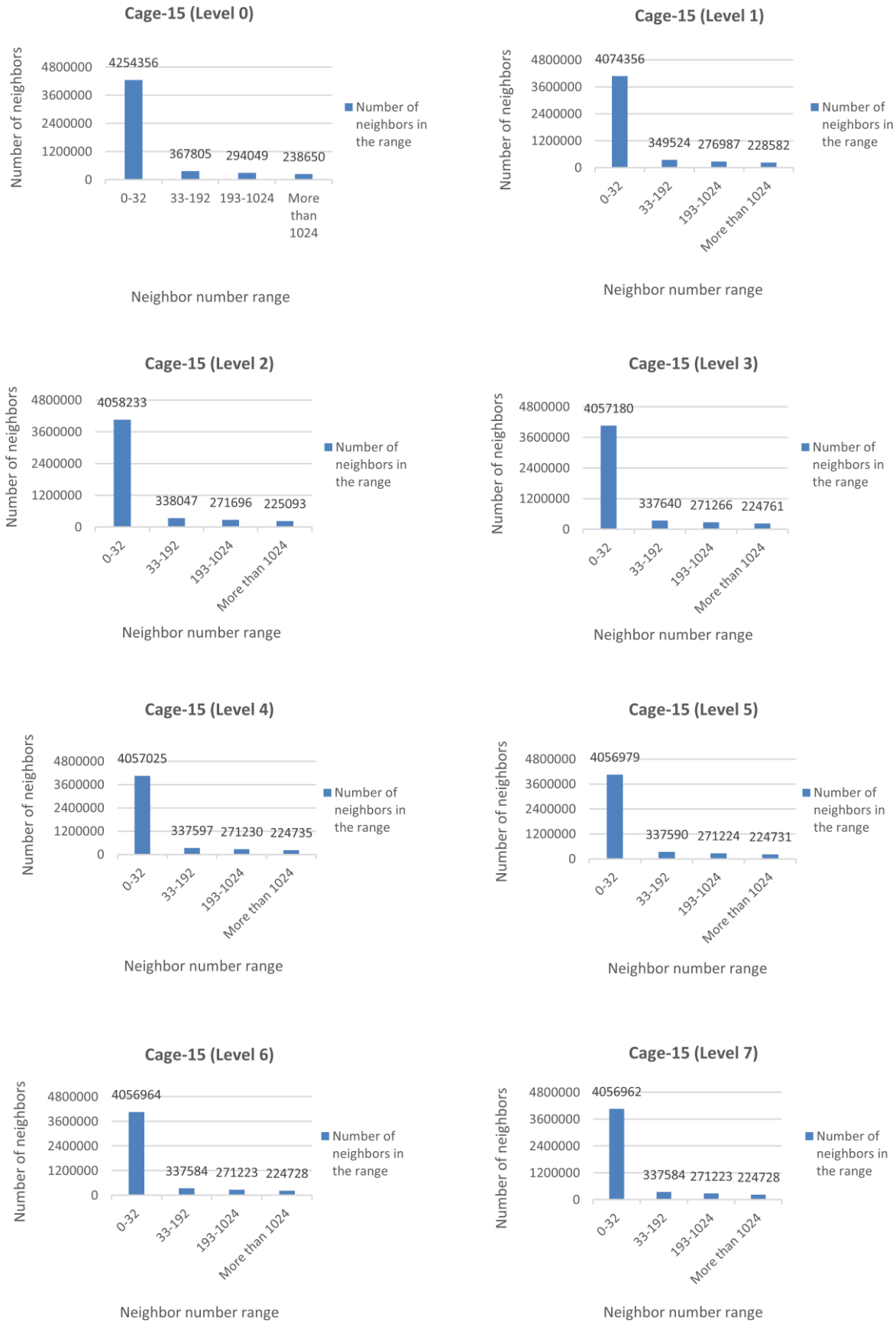

Fig. 2: Classify the number of neighbors of nodes in the Cage-15 benchmark.

Table 1: Platform 1 for examinations

Machine	Specifications
CPU	AMD Opteron(tm) Processor 6344
GPU	NVIDIA Tesla K20Xm
RAM	128 GB
OS	Linux: CentOS release 6.3 (Final)
Compiler	- nvcc: NVIDIA (R) CUDA compiler driver - gcc 4.8.2
CUDA compilation tools	release 6.5, V6.5.12

Table 2: Platform 2 for examinations

device	specifications
CPU	Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz
GPU	Nvidia Geforce GTX 1080 ti
RAM	64 GB
OS	Linux: Ubuntu server 18.04 LTS
Compiler	- nvcc: NVIDIA (R) CUDA compiler driver - gcc version 7.4.0
CUDA compilation tools	release 10.0, V10.0.130

Table 3: Characteristics of the benchmark graphs

Specifications / Benchmark	CNR-2000	EU-2005	IN-2004	Cage-15
number of Nodes	325557	862664	1382908	5154859
number of Links	2,738,969	16,138,468	13591473	47022346
average degree	9.879	22.297	12.233	18.24
maximum in degree	18235	68922	21866	46
maximum out degree	2716	6985	7753	-

Table 4: Execution time (Second) in platform 1

benchmarks / Algorithms	PLM	APLM	NAIM	ACLM	DCLM
CNR-2000	2.74	0.15	0.4	0.10	0.09
EU-2005	3.69	0.49	1.32	0.31	0.28
IN-2004	3.71	0.60	0.95	0.47	0.42
Cage-15	36.88	10.17	13.1	8.32	7.07

Table 5: Execution time (Second) in platform 2

benchmarks / Algorithms	PLM	APLM	NAIM	ACLM	DCLM
CNR-2000	0.78	0.12	0.11	0.09	0.079
EU-2005	2.1	0.36	0.34	0.28	0.25
IN-2004	1.65	0.49	0.42	0.36	0.31
Cage-15	14.52	9.14	---	8.01	6.79

Table 6: Modularity in platform 1

benchmarks / Algorithms	PLM	APLM	NAIM	ACLM	DCLM
CNR-2000	0.912728	0.912667	0.912909	0.912807	0.912807
EU-2005	0.938786	0.937441	0.936152	0.937365	0.937365
IN-2004	0.98031	0.980058	0.978909	0.979802	0.979802
Cage-15	0.893829	0.866638	0.850144	0.866638	0.866638

Table 7: Modularity in Platform 2

benchmarks / Algorithms	PLM	APLM	NAIM	ACLM	DCLM
CNR-2000	0.912812	0.912667	0.912909	0.912807	0.912807
EU-2005	0.938533	0.937441	0.936152	0.937365	0.937365
IN-2004	0.980364	0.980058	0.978909	0.979802	0.979802
Cage-15	0.893814	0.866638	---	0.866638	0.866638

References

[1] M. Guendouz, A. Amine, R. M. Hamou, "Discrete modified fireworks algorithm for community detection in complex networks," Appl. Intell., 46: 373–385, 2017.

[2] D. Sudhakaran, S. Renjith, "Survey of community detection algorithms to identify the best community in real-time networks," Int. J. Sci. Eng. Appl. Sci., (IJSEAS), 2: 529-533, 2016.

[3] S. Fortunato, "Community detection in graphs," Phys. Rep., 486: 75-174, 2010.

[4] M.E.J. Newman, M. Girvan, "Finding and evaluating community structure in networks," Phys. Rev. E, 69: 026113, 2004.

[5] A. Clauset, M.E.J. Newman, C. Moore, "Finding community structure in very large networks," Phys. Rev. E, 70: 066111, 2004.

[6] U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, D. Wagner, "On modularity clustering," IEEE Trans. Knowl. Data Eng., 20: 172-188, 2008.

[7] M. Faysal, S. Arifuzzaman, "Distributed community detection in large networks using an information-theoretic approach," in Proc. 2019 IEEE International Conference on Big Data (Big Data): 4773-4782, 2019.

[8] Q. Ni, J. Guo, W. Wu, C. Huang, "Continuous inuence-based community partition for social networks," arXiv:2002.08554v1 [cs.SI] 20 Feb 2020, 2020.

[9] V.D. Blondel, J.L. Guillaume, R. Lambiotte, E. Lefebvre, "Fast unfolding of communities in large networks," J. Stat. Mech., 10: P10008, 2008.

[10] E. Moradi, M. Fazlali, H. Tabatabaee Malazi, "Fast parallel community detection algorithm based on modularity," in proc. 2015 18th CSI International Symposium on Computer Architecture and Digital Systems (CADS), IEEE, 2015.

[11] C.L. Staudt, H. Meyerhenke, "Engineering parallel algorithms for community detection in massive networks," in proc. 42nd International Conference on Parallel Processing, 2013.

[12] C.Y. Cheong, H.P. Huynh, D. Lo, R.S.M. Goh, "Hierarchical parallel algorithm for modularity-based community detection using GPUs," in proc. 19th International Conference on Parallel Processing, Euro-Par'13: 775–787, 2013.

[13] H. Lu, M. Halappanavar, A. Kalyanaraman, "Parallel heuristics for scalable community detection," Parallel Comput., 47: 9–37, 2015.

[14] M. Fazlali, E. Moradi, H. Tabatabaee Malazi, "Adaptive parallel Louvain community detection on a multicore platform," Microprocess. Microsyst., 54: 26–34, 2017.

[15] J. Zeng, H. Yu, "A scalable distributed louvain algorithm for large-scale graph community detection," in proc. 2018 IEEE International Conference on Cluster Computing (CLUSTER), 2018.

[16] R. Forster, "Louvain community detection with parallel heuristics on GPUs," presented at the IEEE 20th Jubilee International Conference on Intelligent Engineering Systems (INES), Budapest, Hungary, 2016.

[17] L. Zhang, M. Wahib, H. Zhang, S. Matsuoka, "A study of single and multi-device ynchronization methods in nvidia GPUs," in proc. IEEE International Parallel & Distributed Processing Symposium 2020.

[18] Y. Wang, M. Guo, Y. Zhao, J. Jiang, "GPUs‐RRTMG_LW: high-efficient and scalable computing for a longwave radiative transfer model on multiple GPUs," J. Supercomput. 77: 4698–4717, 2020.

[19] M.E.J. Newman, "Modularity and community structure in networks," in proc. Natl. Acad. Sci, 103: 8577–8582, 2006.

[20] D. LaSalle, G. Karypis, "Multi-threaded modularity based graph clustering using the multilevel paradigm," J. Parallel Distrib. Comput., 76: 66-80, 2015.

[21] Y. Guo, Z. Huang, Y. Kong, Q. Wang, "Modularity and mutual information in networks: two sides of the same coin," arXiv preprint arXiv:2103.02542, 2021.

[22] C.L. Staudt, H. Meyerhenke, "Engineering parallel algorithms for community detection in massive networks," IEEE Trans. Parallel Distrib. Syst., 27: 171–184, 2016.

[23] U.N. Raghavan, R. Albert, S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," Phys. Rev. E, 76: 036106, 2007.

[24] G.S. Carnivali, A.B. Vieira, A. Ziviani, P.A.A. Esquef, "CoVeC: Coarse-Grained vertex clustering for efficient community detection in sparse complex networks," Inf. Sci., 522: 180-192, 2020.

[25] X. Que, F. Checconi, F. Petrini, J. Gunnels, "Scalable community detection with louvain algorithm," in proc. 2015 IEEE 29th International Parallel and Distributed Processing Symposium, 2015.

[26] J. Zeng, H. Yu, "Effectively unified optimization for large-scale graph community detection," in proc. 2019 IEEE International Conference on Big Data (Big Data): 475-482, 2019.

[27] W. Fang, X. Wang, L. Liu, Z. Wu, S. Tang, Z. Zheng, "Community detection through vector-label propagation algorithms," arXiv preprint arXiv:2011.08342, 2020.

[28] R.P. Sarmento, "Density-based community detection/optimization" arXiv preprint arXiv: 1904.12593, 2019.

[29] J. Huang, T. Zhang, W. Yu, J. Zhu, E. Cai, "Community detection based on modularized deep nonnegative matrix factorization," Int. J. Pattern Recognit Artif Intell., 35(2): 21590061-215900617, 2020.

[30] S. Souravlas, A. Sifaleras, S. Katsavounis, "Hybrid CPU-GPU community detection in weighted networks," IEEE Access, 8: 57527 – 57551, 2020.

[31] J. Shao, Z. Han, Q. Yang, T. Zhou, "Community detection based on distance dynamics," in proc. 21th ACM SIGKDD international conference on knowledge discovery and data mining. New York: ACM: 1075-1084, 2015.

[32] J. Zhu, X. Ren, P. Ma, K. Gao, "Community detection on complex networks based on a new centrality indicator and a new modularity function," arXiv preprint arXiv:2003.13609, 2020.

[33] I. Gutiérrez, D. Gómez, J. Castro, R. Espínola, "A new community detection algorithm based on fuzzy measures," in Proc. International Conference on Intelligent and Fuzzy Systems: 133-140, 2019.

[34] P. Miasnikof, A.Y. Shestopaloff, A.J. Bonner, "A density-based statistical analysis of graph clustering algorithm performance," J. Complex Networks, 8(3): 1-33, 2020.

[35] D. Jin, B. Zhang, Y. Song, D. He, Z. Feng, S. Chen, W. Li, K. Musial, "ModMRF: A modularity-based Markov Random Field method for community detection," Neurocomputing, 405: 218-228, 2020.

[36] G. Konstantinos, M. Christos, P. Georgios, "A distributed hybrid community detection methodology for social networks," Algorithms, 12(8): 175, 2019.

[37] J.O. Palacio-Niño, F. Berzal "On the use of local structural properties for improving the efficiency of hierarchical community detection methods," arXiv preprint arXiv:2009.06798, 2020.

[38] A. Mahabadi, M. Hosseini, "SLPA-based parallel overlapping community detection approach in large complex social networks," Multimed. Tool. Appl., 80: 6567–6598, 2021.

[39] M.d. Naim, F. Manne, M. Halappanavar, A. Tumeo, "Community detection on the GPU," in proc. 2017 IEEE International Parallel and Distributed Processing Symposium, 2017.

[40] M. Mohammadi, M. Fazlali, M. Hosseinzadeh, "Accelerating Louvain community detection algorithm on graphic processing unit," J. Supercomput., 77: 6056–6077, 2021.

[41] M.E.J. Newman, "Fast algorithm for detecting community structure in networks," Phys. Rev. E, 69: 066133, 2004.

Biographies

Maryam Mohammadi received M.Sc. degrees in Computer Engineering from Science and Research Branch, Islamic Azad University, Kerman, Iran in 2013. She is currently pursuing the Ph.D. degree in the Faculty of Electrical and Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran. Her research interests include parallel processing, hardware implementation, and residue number systems.

Mahmood Fazlali received B.Sc. in Computer Engineering from Shahid Beheshti University (SBU) in 2001. Then he received M.Sc. from University of Isfahan in 2004, and PhD from SBU in 2010 in Computer Architecture. He performed researches on reconfigurable computing systems in computer engineering lab at Technical University of Delft (TUDelft) as a postdoc researcher. Now, he is working as assistant professor at department data and computer sciences at SBU. His research interest includes big data processing, parallel computing and data science.

Mehdi HosseinZadeh received his B.Sc. degree in computer hardware engineering, from Islamic Azad University, Dezfol branch, Iran in 2003. He also received his M.Sc. and the Ph.D. degree in computer system architecture from the Science and Research Branch, Islamic Azad University, Tehran, Iran in 2005 and 2008, respectively. He is currently an Associate professor in Iran University of Medical Sciences (IUMS), Tehran, Iran. He is the author/co-author of more than 120 publications in technical journals and conferences, and his research interests include SDN, Information Technology, Data Mining, Big data analytics, E-Commerce, E-Marketing, and Social Networks.

How to cite this paper:
M. Mohammadi, M. Fazlali, M. Hosseinzadeh, "Parallel louvain community detection algorithm based on dynamic thread assignment on graphic processing unit," J. Electr. Comput. Eng. Innovations, 10(1): 75-88, 2022.

DOI: 10.22061/JECEI.2021.7771.432

URL: https://jecei.sru.ac.ir/article_1565.html