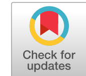
Journal of Parallel and Distributed Computing 132 (2019) 8–20

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

HyPar: A divide-and-conquer model for hybrid CPU–GPU graph processing



Rintu Panja, Sathish S. Vadhiyar *

Department of Computational and Data Sciences, Indian Institute of Science, Bangalore 560012, India

HIGHLIGHTS

- HyPar is a novel model for graph processing on hybrid CPU–GPU architectures.
- It is a divide-and-conquer model with API and runtime strategies.
- Can make use of GPUs even for large graphs that cannot be accommodated in the GPUs.
- Demonstrated with important graph applications including community detection.
- Provides better performance than the prevalent BSP models of executions.

ARTICLE INFO

Article history:
Received 11 January 2018
Received in revised form 15 May 2019
Accepted 26 May 2019
Available online 4 June 2019

Keywords:
Graph algorithms
Hybrid CPU–GPU
Divide-and-conquer

ABSTRACT

Efficient processing of graph applications on heterogeneous CPU–GPU systems require effectively harnessing the combined power of both the CPU and GPU devices. This paper presents *HyPar*, a divide-and-conquer model for processing graph applications on hybrid CPU–GPU systems. Our strategy partitions the given graph across the devices and performs simultaneous independent computations on both the devices. The model provides a simple and generic API, supported with efficient runtime strategies for hybrid executions. The divide-and-conquer model is demonstrated with five graph applications and using experiments with these applications on a heterogeneous system it is shown that our HyPar strategy provides equivalent performance to the state-of-art, optimized CPU-only and GPU-only implementations of the corresponding applications. When compared to the prevalent BSP approach for multi-device executions of graphs, our HyPar method yields 74%–92% average performance improvements.

© 2019 Elsevier Inc. All rights reserved.

1. Introduction

Graph processing has been prevalent in recent years since graph algorithms and abstractions are frequently used to perform analysis in diverse networks such as social, transportation and biological networks. Real world networks are often very large in size resulting in graphs with several hundreds of thousands to billions of vertices and edges. Processing such large-scale graphs is challenging, and many frameworks and algorithms for graph processing have been developed for CPU [11,18,26,29,30,33] and GPU [3,4,20,25,31,36] architectures. In a heterogeneous system consisting of a CPU and GPU, these works utilize only one of the devices for the actual processing. Also, the GPU-only strategies, while providing high performance for small graphs, are limited in terms of exploring large graphs due to the limited memory available on GPU. A hybrid strategy involving computations on both the CPU and GPU cores can help to explore large graphs and utilize all the resources.

There are a limited number of such hybrid frameworks for graph processing that attempt to utilize both the devices [9,15,35]. These efforts employ a Bulk Synchronous Processing (BSP) model across the devices. The BSP model causes communications and synchronizations between the devices at the end of each high level iteration. This causes large communication times and under-utilization of the devices.

This paper presents *HyPar*, a novel programming and runtime model with an API for hybrid CPU-GPU executions of graph applications using Divide-and-Conquer (DC) approach. To solve a problem with the DC approach, the HyPar model employs the strategy of partitioning the graph into two parts for the CPU and GPU, and invoking the original graph problem on the two devices for completely independent processing. The individual results on the two devices are then merged and post processed. HyPar is supported with efficient hybrid runtime strategies and kernel optimizations including automatic determination of the

* Corresponding author.
E-mail addresses: rintupanja@iisc.ac.in (R. Panja), vss@iisc.ac.in (S.S. Vadhiyar).

https://doi.org/10.1016/j.jpdc.2019.05.014
0743-7315/© 2019 Elsevier Inc. All rights reserved.

ratio for partitioning, termination of the independent computations on the devices based on diminishing benefits, efficient modification of graph data structures during merging, and recursive invocation of the steps. Different kernel optimizations are also employed including hierarchical processing of graphs with power-law distributions of vertex degrees and minimizing atomic accesses.

Our divide-and-conquer model is demonstrated with five graph applications, namely, Boruvka's Minimum Spanning Tree, label propagation algorithm for community detection, graph coloring, triangle counting and connected components. Our experiments with these four applications on a heterogeneous systems show that our HyPar strategy provides equivalent performance to the state-of-art, optimized CPU-only and GPU-only implementations of the corresponding applications, achieving up to 98% performance improvement. HyPar is also shown to harness the power of GPUs for large graphs that cannot be entirely accommodated in the GPUs, and hence cannot be executed by the GPU-only implementations. When compared to the prevalent BSP (Bulk Synchronous Processing) approach for multi-device executions of graphs, the divide-and-conquer model followed in HyPar yields 74%–92% average performance improvements. HyPar also provides up to 90% performance improvement over existing multi-core and many-core graph processing frameworks.

2. Related work

Galois [26] is a system for multi-core environments that incorporates the concept of the operator formulation model in which an algorithm is expressed in terms of its action (or operator) on data structures. It has been used to provide large-scale performance for many graph based algorithms. Ligra [29] uses an edge-centric approach which dynamically switches between push and pull method inspired by hybrid BFS algorithm [2]. The framework performs dynamic switching between the sparse and dense representations for the graphs, and also abstracts out the internal traversal details from the user. X-Stream [28] is an edge-centric graph processing framework for in-core as well as out-of-core processing on a single shared memory system. Polymer [34] has shown improved performance over the existing approaches by using NUMA-aware computation. It groups the available cores of a single node and partitions to maximize the accesses to the local memory. Green-Marl [18] provides high-level constructs for users to describe their algorithm intuitively and automatically explores data-parallelism. All of these frameworks have only explored shared memory multi-core CPU systems and do not support GPUs.

Some frameworks including Lonestar-GPU [25], Medusa [36], Gunrock [32] and Groute [3] support GPU-only executions of graph applications. Medusa [36] is a graph processing engine for multi-GPU environment which uses similar BSP (Bulk Synchronous Parallelism) model for communication across GPUs with high-level user interface. The BSP model organizes the computations into super-steps, and involves communications and synchronizations across the devices for every super-step. These frameworks use CPU primarily for reading the graph inputs, coordinating with the GPUs and transferring data to/from GPU. This results in under utilization of the CPU resources. While these GPU-only solutions can potentially provide high performance, they are limited by the sizes of the graphs that can be processed due to limited GPU memory.

To our knowledge, TOTEM [15], Falcon [9] and GGraph [35] are the only frameworks for hybrid CPU-GPU execution of graph applications. They have reported equivalent performance for some of the applications. However, all of these approaches follow BSP model across both the CPU and GPU devices. The communication and synchronization overheads in the BSP model are prominent in heterogeneous systems.

3. HyPar-API

HyPar follows divide-and-conquer approach to solve an application in heterogeneous environment. HyPar first divides the input graph into two parts, for the CPU and GPU devices. Each device then independently solves the problem without any communication with the other device. Then a merge or combine step gathers the result from independent computations. Finally, a post-processing step realizes the remaining computation due to partitioning. For each of these four steps, an API function is provided. The API functions are shown in Table 1.

3.1. Partitioning the graph

The *partGraph* function divides the input graph into two parts one for the CPU and the other for the GPU based on the proportional performance of the given application for the given graph for these two devices. A simple 1-D vertex-block partitioning is used in which the CSR (Compressed Sparse Row) arrays representing the graph are divided into two contiguous segments of vertices along with the edges incident on the vertices.

3.2. Independent computations

After partitioning the graph, our strategy sends the respective parts to the two processing units. The *indComp* function then executes the application independently on the two devices without any communication between the devices. One of the CPU threads, denoted as *GPUdriverThread*, is assigned to drive the GPU execution, and the other CPU threads, denoted as *processingThreads*, for executing the CPU multi-core version. The outputs of this step are the results formed on the two devices represented as arrays. For example, in MST (Minimum Spanning Tree), the component IDs of the vertices formed on a device are stored in an array for the device.

The *indComp* function also has an optional boolean *excpCond* argument. Note that the independent computation on a device involves execution of a graph application/algorithm like BFS, MST etc. on a part assigned to the device. However, execution of the original graph algorithm as such while treating the part as the complete graph input needed by the algorithm will lead to incorrect results. The original algorithm has to be modified such that certain edges or vertices of the part subgraph are not processed while performing the steps of the algorithm. This is enabled by the *excpCond* argument that specifies an exception condition. For example, an exception condition of *EXCPT_BORDER_VERTEX* specifies that the algorithmic steps should not be performed for the border vertices of the part. Our API also provides *EXCPT_BORDER_EDGE* exception condition.

3.3. Merge

This is the step that involves CPU-GPU communication needed for information flow across the cut-edges. The *mergeParts* function merges the results obtained on the two devices due to the independent computations. This step copies the required information from the device to the host and merges to an internal data structure. After merging, the graph data structure in the CPU is updated using optimized parallel graph update routines implemented in our work, explained in the next section. For example, in MST each component is contracted to a single vertex and remove all the internal edges of the component using parallel thread operations. After updating the data structure in the merge step, our strategy decides whether to solve the problem recursively using the previous steps again or to go to the next step depending on the remaining data size.

Table 1
HyPar-API functions.

Function	Remarks
partGraph(*appName*, *graph*)	Partitions the graph into two parts, one for the CPU and another for the GPU.
indComp(*appName*, *graph*, *indCompResult*1, *indCompResult*2, *excpCond*)	Performs independent computations of a graph kernel, given by *appName*, on the two parts. Returns the result in *indCompResult*.
mergeParts(*appName*, *graph*, *indCompResult*1, *indCompResult*2, *mergeResult*)	Merges the results from the independent computations on the two devices into *mergeResult* and updates the graph data structure.
postProcess(*postProcessKernelName*, *graph*, *mergeResult*, *finalOutput*)	Performs post-processing by executing the kernel given by *postProcessKernelName* with the remaining graph.

3.4. Post processing

After the merge step, the algorithm given by the *postProcessKernelName* is run on one of the devices using the remaining data. Our run-time strategy automatically chooses the device for the post processing step at runtime depending on performance on the previous data sets. The final output is made available in the CPU in the *finalOutput* argument.

4. HyPar-runtime optimizations

Several optimizations have been employed in HyPar for realizing the APIs for efficient utilization of heterogeneous processors. Some of the optimizations are automatically executed as runtime strategies while some optimizations are made available as routines to update graph data structures. These routines are for subgraph formation, multi-edge removal, and formation of oriented graphs, and can be utilized for implementing new applications with HyPar.

4.1. Ratio for graph partitioning

To determine the ratio of CPU-GPU performance, a small number of different induced subgraphs (for our study, 3 subgraphs are used) is formed, the original application is executed with each subgraph on both CPU and GPU, the performance ratio is found, and an average of the ratios is obtained for these subgraphs. Each subgraph is generated randomly such that the number of vertices in the subgraph is 5%–10% of the total number of vertices in the original graph. In addition to performance, the GPU memory requirements are also considered to determine the ratio.

4.2. Threshold for independent computations

The CPU-GPU independent computations are performed over several iterations. In some applications, the size of the problem used for the independent computations decreases with the iterations. For example, the number of components in the MST application, the number of nodes with conflicting colors in the graph coloring problem, and the number of nodes with the changed labels in the community detection application, all decrease over time. After a certain threshold, it is advantageous to stop the independent computations and proceed with the merging step since, after this threshold, independent computations may impact the performance due to the lack of sufficient parallelism on both the devices.

Our HyPar-runtime automatically detects this threshold by observing the trend in execution times of the independent computations over multiple iterations. When the execution time does not show further decrease, the runtime automatically switches to perform the merging step.

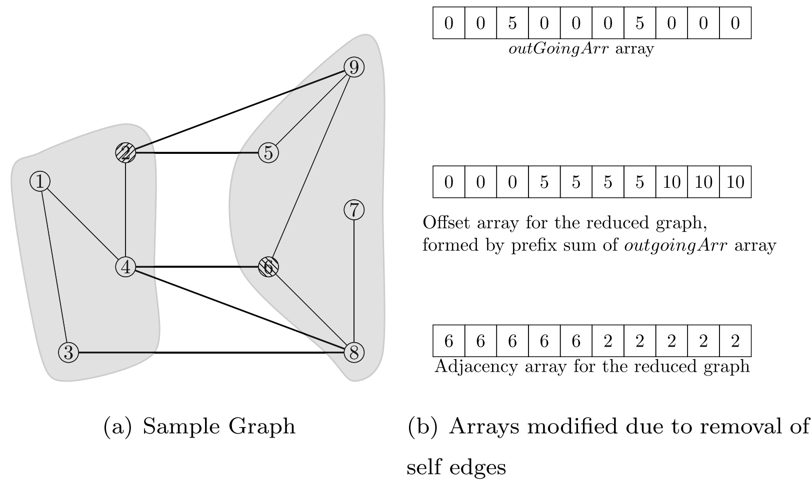
4.3. Parallel self-edge removal and modification of graph data structures

After independent computations, the original graph data structures need to be modified for better memory utilization for subsequent computations. For applications like MST and community detection, independent computations result in the formation of components. For subsequent steps, these components form a reduced graph with the components as vertices and inter-component edges as edges of the reduced graph. Hence, the subsequent steps need to process only the inter-component edges, and not the intra-component edges. A library call and an efficient parallel strategy in the runtime are provided for this phase to remove the intra-component edges, a.k.a *self edges* and modify the graph data structures to represent the reduced graph. This coarsening is done in-place, replacing the original graph data structure, to be able to handle large graphs. The algorithm is explained in Algorithm 1.

Algorithm 1 Parallel Self-edges Removal

```
1: outGoingArr[] = 0
2: for all v ∈ current_set do                          ▷ In Parallel
3:     parent = parent(v)
4:     outEdges = 0
5:     for all u ∈ neighbor(v) do
6:         if parent(u) ≠ parent(v) then
7:             outEdges + +
8:         end if
9:     end for
10:    outEdgeArr[v] = outEdges
11:    atomicUpdate(outGoingArr[parent],outEdges)
12: end for
13: offsetArr = prefixSum(outGoingArr)
14: ∀c ∈ C, startPos[c] = offsetArr[parent(c)]
15: for all v ∈ current_set do                         ▷ In Parallel
16:    parent = parent(v)
17:    outEdges = outEdgeArr[v]
18:    if outEdges > 0 then
19:        storedLoc = atomicAdd(startPos, outEdges)
20:        for all u ∈ adjacent to v do
21:            if parent(u) ≠ parent(v) then
22:                update(adjArr, wtArr)
23:            end if
24:        end for
25:    end if
26: end for
```

CSR representation is followed for the graphs, consisting of the offset, adjacency and weights arrays. For each component, one of the vertices is chosen as a representative. This representative vertex is called as parent vertex of the vertices in the component. For the original graph, each CPU/GPU thread processing a vertex first counts the number of inter-component edges incident with the vertex. The thread then atomically adds this number to a global variable that stores the total number of inter-component edges from a component (*out-edges*). To form a reduced graph, an *outGoingArr* array, of size (number of vertices + 1), is formed in which the elements corresponding to the parent vertices store the total number of out-edges for their components while the other elements are set to 0. This is performed using an atomic update



(a) Sample Graph (b) Arrays modified due to removal of self edges

Fig. 1. Illustration of self-edge removal.

as shown in line 11. Fig. 1(a) shows an example graph in which vertices 2 and 6 are the parents of the two components. Fig. 1(b) shows the *outGoingArr* array.

For the reduced graph, while smaller-sized adjacency and weights arrays are explicitly formed, the size of the offset array is not explicitly reduced. The offset array for the reduced graph is formed by performing an exclusive prefix sum of the *outGoingArr*, as shown in line 13. Fig. 1(b) also shows the offset array for the reduced graph for the example. Then the threads processing the vertices update the adjacency and weights in parallel. This is shown in lines 14–26 of the algorithm. The threads processing the vertices use a global *startPos* variable per component, which maintains the offset at which a thread updates the adjacency and weight arrays. The *startPos* is initialized with the offset of the parent vertex from the offset array (line 14). A thread with out-going edge(s) atomically obtains the old value of *startPos* and increments this variable with the number of its out-edges. The thread then uses the old value as its offset for writing the adjacency and weights values for all its out-edges (line 22). This way, the update of the adjacency and weights arrays for the reduced graph is distributed among all the threads with minimum synchronization. Fig. 1(b) also shows the adjacency array for the reduced graph.

4.4. Recursive invocation of partitioning-independent computations-merging

After merging, the graph data structure is updated and only the required vertices and their outgoing edges are retained. Using experiments it was found that if the reduced graph after the merge step is sufficiently large, it is beneficial to invoke HyPar again using the reduced graph. Our HyPar-runtime follows this recursive approach by again partitioning the reduced graph using already calculated partitioning ratio and performing the *indComp* and *mergeParts* steps. For our current work, the number of edges in the reduced graph (specifically, a threshold of 100 million edges) is used to decide to continue with recursion or to move to the post processing step.

4.5. Parallel subgraph formation

As mentioned, induced subgraphs are formed during the partitioning step. In applications like triangle counting, the ghost edges are also needed for a part. In some cases, the vertices along with their outgoing edges are removed. For example, in coloring, all the vertices that are colored correctly in the independent computations step and their incident edges are removed after the merge step to reduce the size of the graph for further computations. Similarly, in community detection, the adjacent edges of the vertices of the large communities are removed. In all these cases, the graph needs to be updated in an efficient manner.

Using the similar method described in Algorithm 1, the vertices that need to be retained in the updated graph are found. Then, by using the similar method of parallel update to *outGoingArr* array in the algorithm, the updated degrees of the vertices are identified. An exclusive prefix sum is then used to find out the offset array and correspondingly update the required edges information from the original graph data structures.

5. Graph kernel optimizations

In addition to the runtime strategies in our hybrid model, different optimizations were also performed in implementation of the graph kernel functions, primarily related to GPU kernels.

Hierarchical Strategy for Processing Adjacency List: Our graph applications involve exploration of the adjacent vertices of a vertex on the GPUs. A single approach for this exploration may not be optimal for all graph topologies. For example, assigning a single thread to a vertex to explore the adjacency will lead to load imbalance and large bottlenecks for power-law graphs that have a small number of vertices with very high degrees and a large number of vertices with small degrees. An optimization, namely, a hierarchical list based approach [24] is employed for the exploration of adjacency vertices.

In this strategy, the adjacency of a high-degree vertex with degree greater than a threshold size (referred to as *CTA size*) are explored using a Co-operative Thread Array (CTA), which is a group of warps as shown in Fig. 2. For the vertices with degree greater than the warp size but less than the CTA size, all the threads in a warp is used for exploring adjacency. For small degree vertices, a small group of threads within a warp (typically 8) is assigned to explore the adjacency of a vertex. This hierarchical strategy leads to better load balancing across SMs and hence improved performance.

Data-driven and Worklist Approach: The GPU computations can be organized as *topology-driven* or *data-driven* computations. In the topology-driven algorithms [16], GPU threads are spawned for all nodes in a graph, while in the data-driven algorithms [25],

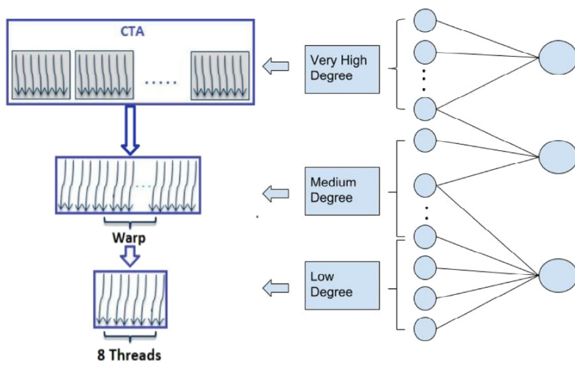


Fig. 2. Co-operative Thread Array (CTA).

worklists are built dynamically and threads are spawned corresponding to only active elements/nodes that have to be processed in a time step. The data-driven approach is used in all our applications.

Reducing Global Atomic Collisions: Many graph operations require atomic accesses to a global memory unit. It is important to reduce these atomic accesses, since atomic constructs serialize the code, are expensive and may cause impact in performance. For our applications, the number of atomic accesses is minimized by batching atomic accesses into a single atomic access and performing hierarchical atomic accesses [13].
Batching of Accesses: In the MST application, each thread that processes a component vertex can merge the vertex with its neighboring component vertex to form a larger component. The thread subsequently has to decrement the number of components that is maintained in a global variable. This access has to be made atomic since multiple threads processing different vertices may decide to decrement the total number of components. We adopt the strategy of first counting the total number of decrementing threads within a warp, and subtracting this number from the global count. This batching of accesses results in the reduced number of atomic accesses. The primitives available in the CUDA library, namely, *__ballot()*, *__popc()* and *__ffs()*, are used to count the number of subtractions within a warp.
Hierarchical Atomics: Atomics are also used in the MST application when choosing a component for merging with another component, C. Each vertex in the component C finds its lightest edge, $E_l(v)$, connecting it to a vertex in another component. The algorithm then chooses the lightest edge, $E_l(C)$, of a component by finding the minimum of the weights of $E_l(v)$s of all the vertices of C and merge C with the component incident with the edge $E_l(C)$. The minimum weight of the edges of the component are found using an atomic min operation. Performing atomic min by all the threads processing vertices of a component can be expensive. A two-level minimum finding approach is used. In the first level, threads processing vertices within a warp that belong to the component C use a warp-level atomicMin to find the warp-level minimum. In the second level, the minimum of all the warp-level minimums are found using a second-level atomicMin operation.

6. Graph applications using HyPar

HyPar follows a divide-and-conquer (DC) approach for hybrid CPU and GPU processing, and hence amenable for applications that follow a DC approach. Five graph algorithms/applications have been implemented using our HyPar divide-and-conquer model. The applications present different levels of complexity. The section begins with a graph coloring application which is easily amenable to DC approach. Then the section explores Boruvka's Minimum Spanning Tree (MST), which is a greedy algorithm and hence may seem not amenable for HyPar's DC approach. However, by making use of the exception condition for independent processing, provided by HyPar, it is shown that it is possible to devise efficient hybrid strategy. Our third application of community detection belongs to the category of a DC application in which the results from the independent computations need to be refined to obtain correct output. To further extend our model, triangle counting application is implemented, where a simple modification to the input data can yield complete independent processing. In this section, four applications are described. Our fifth application, Connected Components (CC) follows a similar approach to MST, and hence not described in this section for brevity.

6.1. Graph coloring

The graph coloring problem is to assign the minimum number of colors (a.k.a., *chromatic number*) to the vertices such that no two adjacent vertices have the same color. Since this is a NP-hard problem, various heuristics have been proposed. One of the approximation algorithms suitable for distributed computing is the one proposed by Gebremedhin and Manne [14].

This algorithm has two iterative phases. In the first phase, each vertex is assigned the minimum consistent (i.e., non-conflicting) color. The processors synchronize and communicate the assignments of colors at the end of each iteration. However, the first phase can result in conflicting colors for the border vertices that are colored simultaneously at the same iteration. The second phase resolves these conflicts by identifying the vertices with conflicting colors and once again invoking the first phase for these vertices. This algorithm is amenable to the DC approach as each device can perform independent coloring followed by a single communication in the end to identify the vertices with conflicting colors. The HyPar CPU version follows a worklist based approach in which the vertices that need to be colored and those with the conflicting colors are added to the worklist. This method has been recently used to find balanced coloring on shared memory architectures [22].

The HyPar algorithm is shown in Algorithm 2. The procedure *partGraph* partitions the graph into two parts, one for CPU and another for GPU. Then the vertices of individual parts are initialized with default colors by *initColors* routine. The *indComp* routine performs the actual computation on the devices to assign colors to the vertices of each part independently without any need of communication. The HyPar runtime automatically stops the independent computations when the execution times of the iterations stops decreasing. The *mergeParts* routine identifies the vertices colored with conflicting colors due to independent processing, and also updates the graph data structure by only retaining these vertices and their incident edges using the parallel graph update routines provided by HyPar. The HyPar runtime then either recursively invokes independent computations followed by merging or proceeds to postProcess step depending on the size of the reduced graph, as mentioned in Section 4.4. The *postProcess* step is performed with the reduced graph to assign colors to the uncolored border vertices.

6.2. Boruvka's MST

Boruvka's algorithm forms minimum spanning tree (MST) by iteratively finding lightest edges from a component and merging two components (or endpoints) connected by a lightest edge. Initially, all the vertices form single-vertex components. In each iteration, for each component, the lightest edge connecting the component with another component is found. The components

Algorithm 2 Hypar Graph Coloring Algorithm

```
1: procedure MERGEPARTS(appName, G, indCompRes1, indCompRes2, mergeResult)
2:     mergeResult ← indCompRes1 ∪ indCompRes2
3:     mergeResult ← Update color of vertices that are colored inconsistently
4:     markVer ← mark vertices colored correctly
5:     G.removeVerEdges(markVer)
6: end procedure
7:
8: procedure COLOR(G)
9:     partRatio ← partGraph("COLOR",G)
10:    initColors(G,cpuColor,gpuColor)              ▷ Initialize default color
11:    indComp("COLOR",G,cpuColor,gpuColor,NULL)
12:    mergeParts("COLOR",G,cpuColor,gpuColor,mergeResult)
13:    if G.size > threshold then
14:        COLOR(G)
15:    else
16:        postProcess("COLOR",G,mergeResult,finalOutput)
17:    end if
18: end procedure
```

that form the end points of the lightest edge are then merged to form larger components. This operation is called *edge contraction*. In the original algorithm, this process is repeated until a single component containing all the vertices are formed. The edges that are contracted across all the iterations constitute the minimum spanning tree (MST). Clearly, the algorithm follows a greedy approach and needs some modification for implementation with the HyPar DC strategy.

For our hybrid algorithm, the graph is partitioned across the CPU and GPU. Boruvka's MST is then performed on each of the parts on the respective devices. The CPU algorithm is based on Galois' implementation [26]. The GPU algorithm uses a worklist based data-driven approach. While performing independent Boruvka's MST algorithm on the CPU and GPU, care must be taken to ensure that such independent computations do not result in incorrect results since the lightest edge from a component in a part can connect to a vertex in the other part, i.e., can be a cut edge. A divide-and-conquer Boruvka's MST algorithm is formulated in which an exception condition is added to the underlying independent Boruvka's MST computation. Specifically, during the iterative process of Boruvka's MST, if the lightest edge from a component is a cut edge, the component is stopped from further expanding and proceed with the other components.

At the end of the independent Boruvka's MST in a part on a device, multiple components are obtained as output. Fig. 3 illustrates the process. Fig. 3(b) shows the components formed after the independent computations for the graph shown in Fig. 3(a). Note that one of the components, *comp 2*, has a single vertex, vertex 6. The lightest edge from vertex 6 is to the other part. Hence it does not merge with *comp 3* in its part. Fig. 3(c) shows the final output for the example graph. The main parts of the algorithm using our API are shown in Algorithm 3.

Algorithm 3 Hypar MST Algorithm

```
1: procedure MERGEPARTS(appName, G, indCompRes1, indCompRes2, mergeResult)
2:     mergeResult ← indCompRes1 ∪ indCompRes2
3:     G.removeSelfEdge(mergeResult)
4: end procedure
5:
6: procedure MST(G)
7:     partRatio ← partGraph("MST",G)
8:     initRep(G,cpuRep,gpuRep)                     ▷ Initialize with vertex id
9:     indComp("MST",G,cpuRep,gpuRep,EXCPT_BORDER_VERTEX)
10:    mergeParts("MST",G,cpuRep,gpuRep,mergeResult)
11:    postProcess("MST",G,mergeResult,finalOutput)
12: end procedure
```

6.3. Community detection

Community detection is an important graph analytical problem and it is widely used in many applications including finding groups in social networks. Communities of vertices are formed for a graph such that the number of intra-community edges is higher when compared to the number of inter-community edges. One of the methods for community detection is using *label propagation* [30]. This is an iterative method in which the vertices are initialized with their own vertex indices as labels, i.e., each vertex is its own community. In each iteration, a vertex, u, obtains the label of its adjacent community to which u has maximum inter-community edges. In case of tie, one of the adjacent communities with the maximum inter-community edges to u is randomly chosen.

The above label propagation algorithm can be parallelized by parallel exploration of the adjacent communities. The parallel algorithm is shown in Algorithm 4. For our parallel CPU and GPU versions, a worklist based approach is used in which active vertices are maintained. Initially, all vertices are designated as active. If a vertex changes its label in the current iteration, the vertex and its neighbors are added to the worklist to refine their communities in the next iteration.

This algorithm is also not amenable to the DC approach, as making a decision for any vertex to add it to any community need community information of all its adjacent vertices. In our HyPar algorithm, the communities formed with the partitioned graphs in the independent computations step are refined in the merge step. Refining or modifying the results output by the independent computations differentiates this application from the previous two applications. The algorithm is illustrated in Algorithm 5.

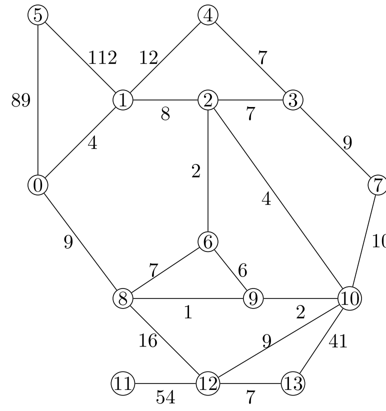
Algorithm 4 Parallel Label Propagation Algorithm

1: Initialize $activeList \leftarrow G.V$
2: **for all** $u \in activeList$ **do** ▷ In Parallel
3: $\quad l \leftarrow argmax(\sum_{v \in N(u)} L(v))$
4: $\quad$ **if** $L(u) \neq l$ **then**
5: $\quad\quad L(u) = l$
6: $\quad\quad activeList \leftarrow activeList \cup N(u)$
7: $\quad$ **else**
8: $\quad\quad activeList \leftarrow activeList/u$
9: $\quad$ **end if**
10: **end for**

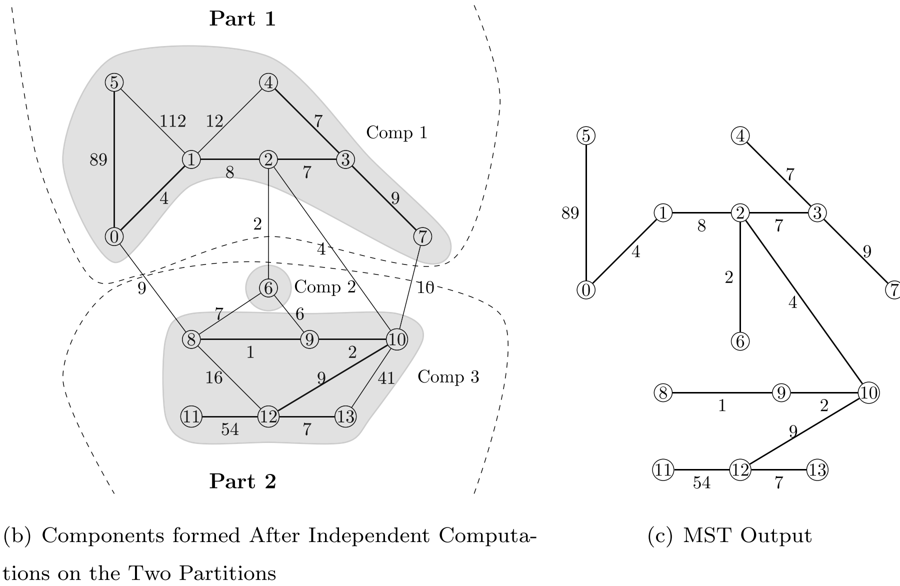
Our HyPar hybrid CPU-GPU implementation performs independent computations of the label propagation algorithm on both the CPU and GPU, and forms local communities in the two devices, with the exception of the border vertices, as indicated by the boolean flag EXCPT_BORDER_VERTEX. During the merging step, the ratio between the intra-community and total number of edges of that community is found and the ratios are averaged across all the communities. Communities with ratios less than the average are dismantled into single-vertex communities using the vertices in these communities. Those communities with very small number of internal edges ($1/10^7$ of the total number of edges) are also dismantled. The other communities are finalized. The graph is then reduced by removing the finalized communities and their incident edges from the graph. The HyPar runtime is then either executed recursively for this reduced graph or post-processed, as described in Section 4.4. The HyPar version is shown in Algorithm 5.

6.4. Triangle counting

Triangle counting is an important application in analyzing the structure of the graph and has many uses including graph clustering. One of the common algorithms in triangle counting is by Arifuzzaman et al. [1] and the subsequent GPU implementation by Adam Polak [27]. The algorithm transforms an undirected graph into an oriented graph. Oriented graph contains only directed edges, where an edge (u, v) is added in the oriented



(a) An Example Graph for MST



(b) Components formed After Independent Computations on the Two Partitions

(c) MST Output

Fig. 3. Illustration of Hybrid MST. (b) shows the three components and the MST edges (thick lines) formed after the independent computations on the two parts for the graph shown in (a). comp 2 consists of a single vertex, vertex 6.

Algorithm 5 Hypar Community Detection Algorithm

```
1: procedure MERGEPARTS(appName, G, indCompRes1, indCompRes2, mergeResult)
2:     mergeResult ← indCompRes1 ∪ indCompRes2
3:     commRatio ← ratio of intra-community and total edges for each community
4:     avgRatio ← Find the average ratio
5:     for v ∈ G.V do                                          ▷ In Parallel
6:         c ← mergeResult[v]
7:         if (commRatio[c] < avgRatio) || (compEdes[c] < minReqEdges) then
8:             mergeResult[v] ← v
9:         else
10:             markVer for removal
11:         end if
12:     end for
13:     G.removeVerEdges(markVer)
14: end procedure
15:
16: procedure COMMDEC(G)
17:     partRatio ← partGraph("COMMDEC",G)
18:     initComm(cpuComm,gpuComm)                    ▷ Initialize with vertex id
19:     indComp("COMMDEC",G,cpuComm,gpuComm,EXCPT_BORDER_VER)
20:     mergeParts("COMMDEC",G,cpuComm,gpuComm, mergeResult)
21:     postProcess("COMMDEC",G,mergeResult,finalOutput)
22: end procedure
```

graph iff $deg(u) < deg(v)$ in the original graph. The algorithm traverses the edges and for each edge (a, b), it finds the triangles containing the edge by finding the intersection of vertices in the adjacency lists of a and b. An undirected graph is transformed to an oriented graph in parallel by using a similar approach to subgraph formation, described in Section 4.5.

The partitioning scheme in our HyPar hybrid version for the triangle counting application is different from the partitioning schemes followed for the other applications. Unlike the other applications, after partitioning, the ghost edges for a part should also be included along with the partition for triangle counting on a device. Then, the triangle counting can proceed independently on both the devices. Unlike the earlier applications, this application contains primarily the independent computations step. The merge/post-processing step is trivial and adds the sums from both the devices.

6.5. Discussion

HyPar is intended for applications amenable for divide-and-conquer executions, approximate applications that do not target high accuracy or those can be formulated as divide-and-conquer applications. There are a significant number of such applications. This paper demonstrates with five such applications. While triangle counting and coloring are amenable for divide-and-conquer style of programming. Label-propagation based community detection is an approximate algorithm that does not necessarily

Table 2
Graph specifications. In the table, M stands for million and B stands for billion.

Graph	$\|V\|$	$\|E\|$	Approx. Diam.	Avg. Deg.	Max. Deg.
road_usa	23.9 M	57.7 M	6262	2.41	9
livejournal_gmembers(lg)	7.48 M	224 M	6	29.99	1,053,749
edit-enwiki(enwiki)	21.5 M	244 M	7	11.35	1,916,963
dbpedia	18.2 M	344 M	9	18.84	632,558
uk-2002	18.5 M	523 M	29	28.27	194,955
R-MAT24	16.8 M	536 M	9	31.9	3,582
eu-2015	11.2 M	759 M	8	67.42	398,609
gsh	30.8 M	1.16 B	9	37.73	2,176,721
arabic	22.7 M	1.26 B	29	55.50	575,662
uk-2005	39.4 M	1.84 B	20	46.69	1,776,858
it-2004	41.2 M	2.27 B	27	55.01	1,326,756

Table 3
1-D block partitioning in HyPar vs state-of-art partitioners in terms of time for partitioning, cut edges and total number of border vertices. M = Million.

Graph	METIS			ParMETIS			HyPar		
	Time (s)	Cut edges	Border nodes	Time (s)	Cut edges	Border nodes	Time (s)	Cut edges	Border nodes
road_usa	24.21	488	488	27	508	506	0.19	0.24 M	1.55 M
lg	118.51	55.34 M	2.78 M	132	53.43 M	2.75 M	0.31	88.59 M	3.62 M
enwiki	251.38	47.32 M	9.92 M	267	43.68 M	9.72 M	0.15	178.55 M	20.53 M
dbpedia	126.64	27.67 M	2.81 M	152	27.46 M	2.80 M	0.23	150.76 M	11.25 M
uk-2002	30.97	1.59 M	0.54 M	73	1.67 M	0.54 M	0.12	13.31 M	2.55 M
eu-2015	71.57	6.04 M	1.16 M	123	6.23 M	1.17 M	0.18	116.71 M	3.41 M
gsh	531.70	77.56 M	10.83 M	989	79.93 M	10.99 M	0.20	148.81 M	18.59 M
arabic	53.10	2.60 M	0.91 M	144	2.12 M	0.82 M	0.14	15.55 M	2.90 M

aim for very strong communities. Boruvka's MST algorithm was formulated into a divide-and-conquer model by using a certain exception condition. The work here encourages application developers to attempt to develop divide-and-conquer models for well-known applications or algorithms. For example, work is in progress to apply divide-and-conquer model for Louvain's community detection by forming communities in independent computations and resolving inconsistencies during the merge step. Such divide-and-conquer models are highly necessary in modern-day multi-device environments. Moreover, the advantage of HyPar is that evolving state-of-art algorithms for CPU and GPU can be plugged in with the hybrid framework.

However, HyPar may not be applicable for inherently incremental and sequential algorithms like BFS, SSSP or PageRank. For these kinds of algorithms, existing BSP models will continue to be the de-facto models for executions. Many of the existing graph frameworks that involve BSP model also apply BSP model of executions to applications amenable for divide-and-conquer models. This paper shows that it is important to develop divide-and-conquer strategies like HyPar for such applications to obtain higher performance than the existing BSP models in multi-device environments. For applications like BFS and SSSP, while existing algorithms may not be able to directly use our model, a fundamental rethink of some of these algorithms may yield a DC formulation. The motivation is that such algorithmic efforts can yield large-scale benefits over the prevalent BSP approaches, as shown in the results

7. Experiments and results

All our experiments were performed on a GPU server consisting of a dual octo-core Intel Xeon E5-2670 2.6 GHz server with CentOS 6.4, 128 GB RAM, and 1 TB hard disk. The CPU is connected to a NVIDIA Tesla K20m GPU card. The K20m GPU has 4.68 GB DDR5 memory, with 2496 core and peak memory bandwidth of 208 GB/s. The CPU portions of our HyPar code were executed with 15 OpenMP threads running on the 15 CPU cores, and one thread maintaining execution of GPU part (GPUdriverThread).

The graphs used in our experiments are shown in Table 2. The graphs were obtained from the University of Florida Sparse Matrix Collection [10], the Laboratory for Web Algorithmics [5,6] and the Koblenz Network Collection [21]. As shown in the table, several real world graphs from different categories and having different characteristics including varying degrees were used for our experiments. These graphs were converted to undirected graphs. GTgraph [23] was used to generate the R-MAT24 graph with parameters $a = 0.5$, $b = c = 0.1$, $d = 0.3$ [8]. For the MST application, random weights were assigned for the edges. All the results shown are obtained using averages of five runs.

7.1. Partitioning

As mentioned earlier, 1-D vertex-block partitioning method is used in HyPar. While existing state-of-art partitioners including METIS, ParMETIS [19] and PaTOH [7] aim to achieve high quality partitioning with minimal number of cut edges, the time taken for partitioning can be large in these tools. The 1-D block partitioning method, on the other hand, can result in a large number of cut edges, but achieve the partitioning in less than a second in most cases due to the simple strategy of partitioning. Table 3 shows the times taken for partitioning and the number of cut edges and total number of border vertices across the two parts by the different partitioners.

For the graphs shown in the table, PaTOH was able to partition only the road_usa graph and not able to partition the other larger sized graphs. For road_usa graph, PaToH takes 24.97 s to form two partitions and yielded 605 cut edges. It is found that the time taken by METIS and ParMETIS are in the range 24 s to 16 min with an average of 3.24 min for some large-scale graphs, while the algorithms that are considered in our work complete execution within a few seconds, as shown in the subsequent sections. Thus, using heavy-weight high quality partitioners like METIS is not applicable to our work. As shown in Table 3, the partitioning time for the 1-D block partitioning in HyPar is less than 0.35 s. It is also found that the number of cut edges by the 1-D block partitioner is very large and can be even 20 times greater than those by METIS. However, the large number of cut edges does not significantly impact the performance of our hybrid algorithms,

Table 4
Workload balance achieved by ratio-based 1D block partitioning. Ratio of execution times of CPU and GPU independent computations for Coloring, MST and Community Detection applications. Ratio=(CPU time)/(GPU time)

Graph	Coloring		MST		CD	
	Ratio-based	50:50	Ratio-based	50:50	Ratio-based	50:50
road_usa	0.60	0.46	0.68	0.52	0.22	0.19
lg	1.01	0.54	0.91	1.62	0.79	0.77
enwiki	0.68	0.60	3.40	3.80	0.82	0.75
dbpedia	0.69	0.50	1.49	0.87	0.98	0.58
uk-2002	0.76	0.52	0.78	1.2	0.64	
R-MAT24	0.81	0.75	0.62	0.58	1.50	
eu-2015	0.98	0.75	1.33		1.93	
gsh	0.77		1.99		1.81	
arabic	0.63		2.09		0.91	

since communications needed for the cut edges happen in only the merging phase and in one batch in our divide-and conquer strategy.

As described, the HyPar runtime performs proportional partitioning of the graph for CPU and GPU based on the ratio of execution times of the application for a few sample graphs on the two devices. This strategy is evaluated in terms of the actual workload balance achieved on the two devices. Table 4 shows the ratio of execution times for the independent computations on the CPU and GPU for coloring, MST and community detection applications for the different graphs, where ratio is obtained as (CPU time)/(GPU time). An ideal partitioning would achieve ratios of 1.0. As shown in the table, HyPar proportional partitioning achieves an average ratio of 1.1 across all graphs and applications. The ratios are the range 0.21–3.39. The wide variation is because the HyPar ratio-based strategy is a best effort quick heuristic based on execution of a limited number of subgraphs on the CPU and GPU. However, compared to a simple 50:50 partitioning, it is found that the ratio-based partitioning achieves a better load balance. It is also found that some graphs could not be executed with the 50:50 partitioning for some applications since the 50% of the graph could not be accommodated on the GPU in those cases.

7.2. Performance improvements for each application

Our HyPar algorithms are compared with state-of-art CPU-only and GPU-only algorithms. In all our experiments, it was verified that the results of HyPar are consistent with the state-of-art algorithms and frameworks for all evaluated algorithms.

7.2.1. Graph coloring

HyPar's hybrid graph coloring implementation is compared with the state-of-art CPU version by Lu et al. [22]. Though their work targets balanced coloring, our comparison is with the first phase of their work that uses first fit coloring algorithm by Gebremedhin and Manne. Comparison is also made with our GPU implementation of the algorithm. In general, GPU version of coloring is slower since the larger number of threads on the GPU yields more conflicting adjacent nodes. It is found that for large graphs, HyPar gives 18%–56%, with an average of 35% performance improvement over the state-of-art CPU version. HyPar also provides 8%–80%, with an average of 42% performance improvement over the GPU version for all the graphs. The large range in benefits due to HyPar is due to the CPU-GPU data transfer overheads that are significant for small graphs, and occupy small percentages in large graphs. It is also verified that the number of colors produced by HyPar for each graph is similar to the state-of-the-art CPU implementation. Table 5 compares the number of colors produced by the different versions.

Table 5
Comparison of number of colors.

Graph	CPU version	GPU version	HyPar
road_usa	4	4	4
lg	43	52	42
enwiki	88	110	88
dbpedia	62	77	60
uk-2002	943	933	943
R-MAT24	1042	1087	1031
eu-2015	9866	9870	9866
gsh	9915		9916
arabic	3247		3247
uk-2005	588		589
it-2004	3221		3221

Table 6
Coloring: Comparison with Deveci et al.'s multi-core CPU version.

Graph	Deveci et al.'s Exec. Time (s)	HyPar's Exec. Time (s)
road_usa	0.37	0.67
lg	1.59	0.72
enwiki	3.60	1.18
dbpedia	3.78	1.45
uk-2002	0.37	1.49
eu-2015	73.67	1.16
gsh	51.49	2.29
arabic	2.91	1.81

HyPar is also compared with the work by Deveci et al. [11]. Comparison was made with their code available in the Kokkos kernel [12]. Table 6 shows the comparison results with their multi-core CPU version. It is found that except for road_usa and uk-2002 graphs, HyPar gives 37%–98% performance improvement over their version. Their multi-core GPU version was able to accommodate and execute only our smallest graph of road_usa. For this graph, the execution times of their code and the HyPar code were 0.37 and 0.67 s, respectively, and thus were comparable.

7.2.2. Boruvka's MST

HyPar's Boruvka's MST is compared with state-of-art CPU (Galois [17]) and GPU (Lonestar-GPU [25]) versions. The results are shown in Fig. 4. With efficient runtime strategies, our HyPar strategy yielded 55%–84% performance improvement over Lonestar-GPU version for MST. Compared to Galois CPU implementation except for the first graph, our HyPar method gives either equivalent or up to 36% improved performance. Also, the first four graphs correspond to the small graphs that can be entirely accommodated on the GPU, while the next seven graphs are large graphs that cannot be entirely accommodated on the GPU. This also demonstrates one of the primary uses of our HyPar algorithm: for graphs that cannot be entirely accommodated on the GPU, our HyPar hybrid strategy attempts to use the power of both the CPU and GPU by appropriate partitioning. HyPar's better performance than the Galois CPU version in some cases is due to the utilization of GPU capacities in HyPar. In some cases, more benefits were found with the HyPar version because for these cases, the independent computations make some large components and thus decrease the remaining graph sizes by large factors.

7.2.3. Community detection

Our HyPar community detection is compared with the state-of-art CPU Label Propagation version by Staudt and Meyerhenke [30] called PLP. Our HyPar algorithm is also based on this CPU version. For uniform comparisons with the work by Staudt and Meyerhenke, a uniform stopping criteria was used in which the executions were performed till the number of active vertices in an iteration is less than $(1/10^5)$ of the total

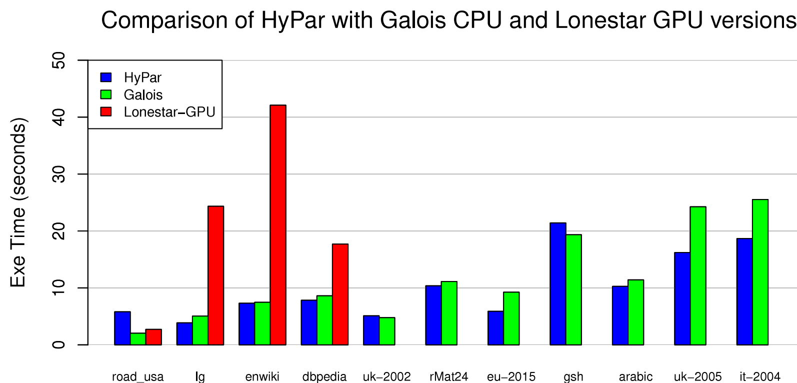


Fig. 4. MST: Comparison of HyPar with state-of-art CPU (Galois) and GPU (Lonestar-GPU) versions.

number of vertices or when the number of iterations reaches 100. Comparisons are also made with the state-of-art PLP algorithm on GPUs by Kozawa et al. [20]. For uniform comparison, same criteria was used for active vertices ($1/10^5$ of the total number of vertices) while the maximum number of iterations was fixed as 10.

Fig. 5 shows the comparison results. The code by Kozawa et al. could not be executed for some of the graphs due to memory limitations on the GPU. When compared to the CPU version by Staudt and Meyerhenke, it is found that for seven of the graphs, our HyPar method provides performance improvement of 15%–60%, with an average of 34%. This is due to the harnessing of the GPU's capabilities. For road_usa, edit-enwiki, uk-2005 and eu-2015, slight slowdowns were obtained. For these graphs, the HyPar algorithm breaks many small communities into single nodes during the merge step (see the algorithm in Section 6). Thus, the complexities of the reduced graphs for the subsequent steps are relatively high. When compared with the GPU version by Kozawa et al. it is found that except for the road_usa graph, their work gives 13%–88% better performance than HyPar. The advantage of HyPar is its generic mechanisms that can be applied to multiple applications. Also, any state-of-art work for a particular device like the algorithm for GPUs by Kozawa et al. can be integrated into HyPar to obtain even better performance by making use of additional device, which in this case is CPU.

Our HyPar's label propagation algorithm makes approximations to the PLP algorithm by Staudt and Meyerhenke [30] due to the independent formation of communities in the CPU and GPU. Hence, the results produced will not be the same as in the CPU-based algorithm. Modularity is one of the metrics that attempts to capture the goodness of the communities formed by a community detection algorithm and is defined as:

$$Q = \sum_{c \in C} [\frac{\sum_{in}^{c}}{2m} - (\frac{\sum_{tot}^{c}}{2m})^2] \quad (1)$$

where m is the total number of edges of the graph, $\sum_{in}^{c}$ is the total number of intra-edges for a community c and $\sum_{tot}^{c}$ is the total number of edges of the community c. Table 7 compares the modularity values of the communities formed in the HyPar, the CPU version by Staudt and Meyerhenke and the GPU version by Kozawa et al. It is found that with our modified algorithm of merging step to dismantle small communities, HyPar's approximation version is giving equivalent modularity values to the other two algorithms.

7.2.4. Triangle counting

HyPar's triangle counting implementation was compared with our parallel CPU and the state-of-art GPU [27] versions. HyPar provides 4%–48%, with an average of 22% performance improvement over the CPU version, and 11%–54%, with an average of 30% performance improvement over the state-of-art GPU version.

HyPar's triangle counting was also compared with the triangle counting algorithm for GPUs by Bisson and Fatica [4]. For this comparison, a system of Intel Haswell CPU processors and NVIDIA K40 GPU was used, HyPar was executed for some of the graphs used in their work, and times are compared with the times reported in their paper on a K40 GPU. Table 8 shows the results. It is found that the algorithm by Bisson and Fatica gives 1.47X–3.11X better performance than HyPar. HyPar is a general tool for hybrid CPU-GPU executions, applicable to multiple applications. The advantage of HyPar is that evolving state-of-art implementations like the triangle counting algorithm by Bisson and Fatica can be used for its GPU executions.

7.3. Comparison with state-of-art graph processing frameworks and BSP models

7.3.1. Comparison with hybrid CPU–GPU framework and BSP models

HyPar's divide-and-conquer hybrid strategy is also compared with the popular BSP (Bulk Synchronous Parallel) model used for multi-device and hybrid executions. In the BSP model, the computations are organized into super-steps corresponding to the outer loop of the original algorithms. At the end of each super-step, necessary communications and synchronizations are performed between multiple devices holding the different parts. Totem [15] is a representative Hybrid CPU-GPU framework for graph applications and uses BSP model for communication across devices.[1]

HyPar and Totem were compared with two applications, namely, connected components (CC) and graph coloring. The graph coloring application was implemented within the Totem framework. Note that triangle counting does not need a BSP model of bulk synchronism due to the completely independent nature of computations in the multiple devices with the final result only needing counts of the triangles produced in the different devices. MST and community detection algorithms could not be implemented with Totem since the two applications required both pull and push methods of inter-device communication in each super-step, which Totem does not provide. Hence, our own BSP implementations of MST and community detection applications were implemented.

Table 9 shows the comparison results in terms of execution times. For both the strategies, the time includes initialization,

[1] The other hybrid CPU-GPU frameworks for graph applications that we are aware of, Falcon [9] and GGraph [35], are shown to produce equivalent results to Totem and also use BSP model.

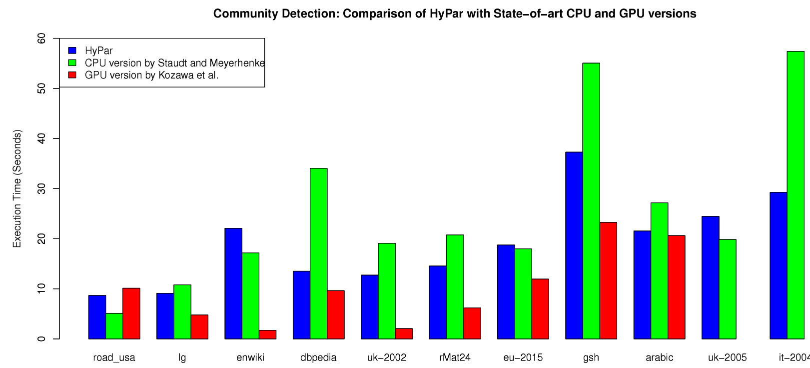


Fig. 5. Community detection: Comparison of HyPar with state-of-art CPU version by Staudt and Meyerhenke and GPU version by Kozawa et al.

Table 7
Community detection: Comparison of modularities of the CPU, GPU and HyPar versions.

Graph	Modularity (Staudt and Meyerhenke)	Modularity (Kozawa et al.)	Modularity (HyPar)
road_usa	0.841	0.748	0.844
lg	0.005	0.12	0.004
enwiki	0.012	0.038	0.012
dbpedia	0.259	0.396	0.139
uk-2002	0.960	0.348	0.964
R-MAT24	0.004	0.001	0.032
eu-2015	0.811	0.808	0.799
gsh	0.573	0.498	0.597
arabic	0.953	0.924	0.957
uk-2005	0.954		0.963
it-2004	0.944		0.957

Table 8
Triangle counting: Comparison with Bisson and Fatica GPU algorithm.

Graph	Execution time by Bisson and Fatica (s)	Execution time by HyPar (s)
mouse_gene	0.55	1.71
soc-livejournal	0.35	0.86
hollywood-2009	1.38	2.33
kron_g500-logn18	0.61	1.25
kron_g500-logn211	1.98	17.58

partitioning and execution time of the applications. For road_usa, Totem was not able to complete its execution. It is found that overall, our HyPar hybrid execution gives the following performance improvements over Totem: 83%–97%, with an average of 92% for coloring and 54%–87%, with an average of 74% for CC. It gives the following performance improvements over our BSP models: 19%–61%, with an average of 48% for MST, and 4%–63%, with an average of 40% for community detection. The large scale improvements due to HyPar shown in these results point to the significant performance impact that our HyPar model can make in hybrid CPU-GPU and multi-device executions of graph applications for which the BSP models are commonly used.

7.3.2. Comparison with multi-core and many-core frameworks

Ligra [29] is one of the state-of-the-art shared memory framework for graph processing which uses edge based parallelism. It also changes between push and pull method for exploration depending on the sum of the degree of the vertices in the frontier set. Comparisons are made with two applications that are present both in Ligra and our work.

Table 10 shows the comparison results in terms of execution times. As shown in the table, for Triangle Counting (TC) application, performance improvement of upto 90%, with an average of 59%, is obtained over Ligra. For Connected Components(CC), up to 67% is obtained, with an average of 24% performance improvement. Results are not shown for the last two graphs since Ligra was not able to complete execution within 1000 s for these graphs.

HyPar's connected component algorithm was also compared with the Gunrock GPU framework [31]. Table 11 shows the comparisons for small graphs. Gunrock gave out-of-memory errors for graphs larger than dbpedia. It is found that except for road_usa, HyPar gives equivalent results to Gunrock. The advantage of HyPar is that it can make use of Gunrock in its independent computations on GPU to obtain even better performance.

7.4. Analysis of phases

Fig. 6 shows the execution times of the different phases of HyPar for three graphs that represent small, medium and large graphs.

The independent computations phase is the primary phase that performs the actual algorithmic tasks. The other three phases correspond to the extra tasks performed by our method to realize the hybrid executions using the partitioning approach. It is found that while these extra tasks occupy significant percentages, about 56%–86%, of the overall time for the small graph, the percentages decrease to 18%–40% for larger problem sizes. Correspondingly, the percentage of the execution time occupied by the actual tasks performed by the independent computations increase with increasing graphs sizes and attain up to 81%. As HyPar performs this step without any communication between the devices, a significant performance improvement is obtained over BSP approach. Among the extra tasks, our quick and efficient 1D-block partitioning approach occupies less than 7% in most cases. The merging and post processing phases occupy significant percentages due to the CPU-GPU data transfers and consolidating the results of the independent computations. Our future work will involve optimizations of these phases.

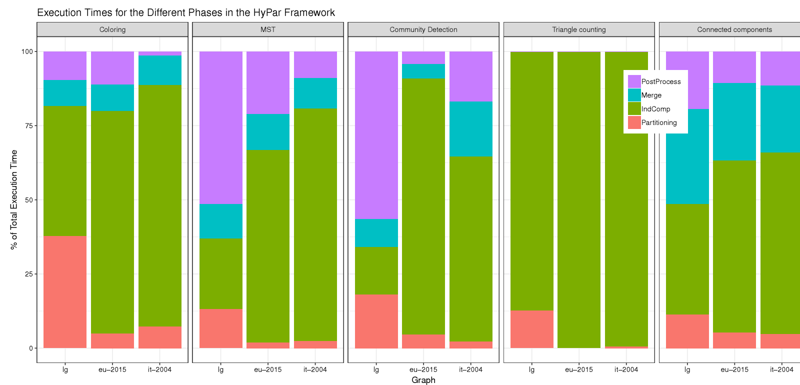


Fig. 6. Execution times of different phases.

Table 9
Performance comparison with Totem and BSP model in terms of execution times.

Graph	Coloring		MST		Community detection		CC	
	Totem (s)	HyPar (s)	Our BSP (s)	HyPar (s)	Our BSP (s)	HyPar (s)	Totem (s)	HyPar (s)
road_usa	7.92	0.66	8.13	5.80	11.63	8.66	–	26.74
lg	10.31	0.72	9.01	3.86	22.66	9.08	7.97	1.76
enwiki	16.87	1.44	18.78	7.30	53.13	22.06	14.53	2.73
dbpedia	14.90	1.18	19.94	7.82	39.96	14.56	14.72	2.70
uk-2002	19.67	1.48	11.23	5.10	24.86	15.18	5.84	2.38
R-MAT24	11.25	1.81	17.95	10.36	20.59	14.55	8.66	3.98
eu-2015	18.48	1.16	10.88	5.88	37.06	19.86	15.46	1.86
gsh	25.87	2.28	40.90	33.12	56.86	37.27	22.43	5.56
arabic	63.70	1.81	23.66	10.98	37.39	23.66	13.12	3.72
uk-2005	51.00	2.37	38.38	16.21	29.60	14.04	25.94	5.42
it-2004	57.53	2.60	40.94	19.63	38.12	36.72	24.77	5.56

Table 10
Performance comparison with Ligra in terms of execution times.

Graph	TC		CC	
	Ligra (s)	HyPar (s)	Ligra (s)	HyPar (s)
road_usa	6.12	1.58	82.4	26.74
lg	14.8	3.74	1.73	1.75
enwiki	18.4	5.40	4.07	2.73
dbpedia	10.6	7.33	3.29	2.69
uk-2002	29.5	3.17	4.44	2.37
R-MAT24	6.17	6.05	3.89	3.98
eu-2015	469	196.43	2.08	1.86
gsh	424	242.31	6.85	5.56
arabic	211	20.64	5.46	3.64
uk-2005	–	14.04	9.34	5.39
it-2004	–	35.55	7.64	5.56

Table 11
Connected components: Comparison with Gunrock.

Graph	Gunrock (s)	HyPar (s)
road_usa	4,61	26.74
lg	2.17	1.75
enwiki	2.48	2.73
dbpedia	2.68	2.7

8. Conclusions and future work

This paper presented *HyPar*, a novel divide-and-conquer model for hybrid CPU-GPU executions of graph applications. Our experiments showed that our HyPar model provides equivalent performance to the state-of-art, optimized CPU-only and GPU-only implementations of the corresponding applications, achieving up to 98% performance improvement. HyPar is also shown to harness the power of GPUs for large graphs that cannot be entirely accommodated in the GPUs, and hence cannot be executed by the GPU-only implementations. HyPar also provides up to 90% performance improvement over existing multi-core and many-core graph processing frameworks. The advantage of HyPar is that it is a generic tool into which evolving state-of-art algorithms for specific devices can be integrated. In future, we plan to extend our model for multi-node and multi-partition executions.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.jpdc.2019.05.014.

References

[1] S. Arifuzzaman, M. Khan, M. Marathe, PATRIC: A parallel algorithm for counting triangles in massive networks, in: 22nd ACM International Conference on Information & Knowledge Management, 2013, pp. 529–538.

[2] S. Beamer, K. Asanović, D. Patterson, Direction-optimizing breadth-first search, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, 2012.
[3] T. Ben-Nun, M. Sutton, S. Pai, K. Pingali, Groute: An asynchronous multi-GPU programming model for irregular computations, in: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2017, pp. 235–248.
[4] M. Bisson, M. Fatica, High performance exact triangle counting on GPUs, IEEE Trans. Parallel Distrib. Syst. 28 (12) (2017) 3501–3510.
[5] P. Boldi, M. Rosa, M. Santini, S. Vigna, Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks, in: S. Srinivasan, K. Ramamritham, A. Kumar, M.P. Ravindra, E. Bertino, R. Kumar (Eds.), Proceedings of the 20th International Conference on World Wide Web, 2011, pp. 587–596.
[6] P. Boldi, S. Vigna, The webgraph framework I: Compression techniques, in: Proc. of the Thirteenth International World Wide Web Conference, WWW 2004, 2004, pp. 595–601.
[7] Ü. Çatalyürek, C. Aykanat, Patoh (partitioning tool for hypergraphs), in: Encyclopedia of Parallel Computing, Springer, 2011, pp. 1479–1487.
[8] D. Chakrabarti, Y. Zhan, C. Faloutsos, R-MAT: A recursive model for graph mining, in: Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004, 2004, pp. 442–446.
[9] U. Cheramangalath, R. Nasre, Y.N. Srikant, Falcon: A graph manipulation language for heterogeneous systems, ACM Trans. Archit. Code Optim. 12 (4) (2015) 54:1–54:27.
[10] T.A. Davis, Y. Hu, The university of florida sparse matrix collection, ACM Trans. Math. Software 38 (1) (2011) 1.
[11] M. Deveci, E.G. Boman, K.D. Devine, S. Rajamanickam, Parallel graph coloring for manycore architectures, in: Parallel and Distributed Processing Symposium, 2016 IEEE International, 2016, pp. 892–901.
[12] C. Edwards, C. Trott, D. Sunderland, Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, J. Parallel Distrib. Comput. 74 (12) (2014) 3202–3216.
[13] I.J. Egielski, J. Huang, E.Z. Zhang, Massive atomics for massive parallelism on GPUs, in: Proceedings of the 2014 International Symposium on Memory Management, ISMM '14, 2014, pp. 93–103.
[14] A. Gebremedhin, F. Manne, Scalable parallel praph coloring algorithms, Concurrency, Pract. Exp. 12 (12) (2000) 1131–1146.
[15] A. Gharaibeh, L.B. Costa, E. Santos-Neto, M. Ripeanu, A Yoke of Oxen and a thousand chickens for heavy lifting graph processing, in: International Conference on Parallel Architectures and Compilation Techniques, PACT '12, Minneapolis, MN, USA - September 19–23, 2012, 2012, pp. 345–354.
[16] P. Harish, P. Narayanan, Accelerating large graph algorithms on the GPU using CUDA, in: International Conference on High-Performance Computing, 2007, pp. 197–208.
[17] M.A. Hassaan, M. Burtscher, K. Pingali, Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms, in: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP '11, 2011, pp. 3–12.
[18] S. Hong, H. Chafi, E. Sedlar, K. Olukotun, Green-marl: a DSL for easy and efficient graph analysis, ACM SIGARCH Comput. Archit. News 40 (1) (2012) 349–362.
[19] G. Karypis, V. Kumar, A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN, 1998.
[20] Y. Kozawa, T. Amagasa, H. Kitagawa, GPU-Accelerated graph clustering via parallel label propagation, in: Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM, 2017, pp. 567–576.
[21] J. Kunegis, Konect: the Koblenz network collection, in: Proceedings of the 22nd International Conference on World Wide Web, 2013, pp. 1343–1350.
[22] H. Lu, M. Halappanavar, D. Chavarría-Miranda, A. Gebremedhin, A. Kalyanaraman, Balanced coloring for parallel computing applications, in: Parallel and Distributed Processing Symposium, IPDPS, 2015 IEEE International, 2015, pp. 7–16.
[23] K. Madduri, D. Bader, GTgraph: A suite of synthetic random graph generators, http://www.cse.psu.edu/~madduri/software/GTgraph.
[24] D. Merrill, M. Garland, A. Grimshaw, Scalable GPU graph traversal, in: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '12, 2012, pp. 117–128.
[25] R. Nasre, M. Burtscher, K. Pingali, Data-driven versus topology-driven irregular computations on GPUs, in: Parallel & Distributed Processing, IPDPS, 2013 IEEE 27th International Symposium on, 2013, pp. 463–474.
[26] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M.A. Hassaan, R. Kaleem, T. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, X. Sui, The tao of parallelism in algorithms, in: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4–8, 2011, 2011, pp. 12–25.
[27] A. Polak, Counting triangles in large graphs on GPU, in: Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International, IEEE, 2016, pp. 740–746.
[28] A. Roy, I. Mihailovic, W. Zwaenepoel, X-stream: Edge-centric graph processing using streaming partitions, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, 2013, pp. 472–488.
[29] J. Shun, G.E. Blelloch, Ligra: A lightweight graph processing framework for shared memory, in: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, 2013, pp. 135–146.
[30] C.L. Staudt, H. Meyerhenke, Engineering high-performance community detection heuristics for massive graphs, in: Parallel Processing, ICPP, 2013 42nd International Conference on, 2013, pp. 180–189.
[31] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, J. Owens, Gunrock: A high-performance graph processing library on the GPU, in: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016, 2016, pp. 11:1–11:12.
[32] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, J.D. Owens, Gunrock: A high-performance graph processing library on the GPU, in: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2016, pp. 11.
[33] M.M. Wolf, M. Deveci, J. Berry, S. Hammond, S. Rajamanickam, Fast linear algebra-based triangle counting with kokkoskernels, in: 2017 IEEE High Performance Extreme Computing Conference, HPEC, 2017, pp. 1–7.
[34] K. Zhang, R. Chen, H. Chen, NUMA-aware graph-structured analytics, in: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, 2015, pp. 183–193.
[35] T. Zhang, J. Zhang, W. Shu, M.-Y. Wu, X. Liang, Efficient graph computation on hybrid CPU and GPU systems, J. Supercomput. 71 (4) (2015) 1563–1586.
[36] J. Zhong, B. He, Medusa: Simplified graph processing on GPUs, IEEE Trans. Parallel Distrib. Syst. 25 (6) (2014) 1543–1552.

Rintu Panja received his B.Tech degree from Institute of Engineering and Management in 2012 and is now doing his masters in Department of Computational and Data Sciences, Indian Institute of Science. His research interests include GPU computing, distributed computing and parallel graph processing.

Sathish S. Vadhiyar is an Associate Professor in the Department of Computational and Data Sciences, Indian Institute of Science. He obtained his B.E. degree from the Department of Computer Science and Engineering at Thiagarajar College of Engineering, India in 1997 and received his Master's degree in Computer Science at Clemson University, USA in 1999. He graduated with a Ph.D. from the Computer Science Department at University of Tennessee, USA in 2003. His research areas are building application frameworks including runtime frameworks for irregular applications, hybrid execution strategies, and programming models for accelerator-based systems, processor allocation, mapping and remapping strategies for networks for different application classes including irregular, multi-physics, climate and weather applications, middleware for production supercomputer systems and fault tolerance for large-scale systems.