MND-MST: A Multi-Node Multi-Device Parallel Boruvka's MST Algorithm

Rintu Panja
Department of Computational and Data Sciences
Indian Institute of Science
Bangalore, India
rintupanja@iisc.ac.in

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

ABSTRACT

Efficient processing of large-scale graph applications on heterogeneous CPU-GPU systems require effectively harnessing the combined power of both the CPU and GPU devices. Finding minimum spanning tree (MST) is an important graph application and is used in different domains. When applying MST algorithms for large-scale graphs across multiple nodes (or machines), the existing approaches use BSP (bulk synchronous parallel) model involving large-scale communications. In this paper, we propose a multi-node multi-device algorithm for MST, *MND-MST*, that uses a divide-and-conquer approach by partitioning the input graph across multiple nodes and devices and performing independent Boruvka's MST computations on the devices. The results from the different nodes are merged using a novel hybrid merging algorithm that ensures that the combined results on a node never exceeds it memory capacity. The algorithm also simultaneously harnesses both CPU and GPU devices. In our experiments, we show that our proposed algorithm shows 24-88% performance improvements over an existing BSP approach. We also show that the algorithm exhibits almost linear scalability, and the use of GPUs result in upto 23% improvement in performance over multi-node CPU-only performance.

CCS CONCEPTS

• **Computing methodologies** → *Massively parallel algorithms*; Distributed algorithms;

KEYWORDS

Boruvka's MST, Hybrid CPU and GPU, Multi-node and multi-device, Divide and conquer

ACM Reference Format:
Rintu Panja and Sathish Vadhiyar. 2018. MND-MST: A Multi-Node Multi-Device Parallel Boruvka's MST Algorithm. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225146

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 require effectively harnessing the power of multiple nodes and devices.

Finding minimum spanning tree (MST) is one of the important graph applications and is used to solve different problems like very large scale integration design, design of networks and approximating several problems like travelling salesman problem, maximum flow problem, weighted perfect matching problem etc. For finding MST on large graphs, multiple nodes with distributed memory parallelism have to be employed in which the graph is partitioned across the nodes. Existing distributed memory MST algorithms employ a bulk synchronous parallel (BSP) model in which the graph processing is organized into supersteps and the nodes synchronize and communicate at the end of every superstep [12, 17, 20]. Such BSP models can involve heavy communications, thus impacting performance, and can also cause under-utilization of resources. To avoid such frequent communications, graph applications will have to be formulated to independently execute the algorithmic steps in multiple devices. This will enable effectively harnessing the combined power of CPU and GPU in heterogeneous systems and reducing the number of messages across different cluster nodes and devices.

In this paper, we propose a distributed memory multi-node multi-device MST algorithm that follows a divide-and-conquer paradigm. The algorithm first partitions the graph across the compute nodes and further partitions the graph for the CPU and GPU devices in a single node. We invoke Boruvka's MST algorithm on each device for completely independent processing on the devices. The individual results formed on the multiple nodes are then merged using a novel hierarchical merging algorithm. The merging algorithm ensures that at any point, any intermediate merged result formed on a node does not exceed the memory capacity of the node, thus facilitating the exploration of large-scale graphs. Our algorithm also harnesses the combined power of both the CPU cores and GPUs on multiple nodes by simultaneous executions on all the devices. Our algorithm uses our HyPar framework, consisting of an API and runtime strategies for efficient CPU-GPU executions. The runtime strategies include termination of the independent computations on the devices based on diminishing benefits, efficient modification of graph data structures in heterogeneous(CPU and GPU) environments and recursive invocation of the steps. We also employ different kernel optimizations including hierarchical processing of graphs with power-law distributions of vertex degrees and minimizing atomic accesses.

In our experiments, we show that our proposed algorithm shows 24-88% performance improvement over an existing BSP approach, with 40-92% reduction in communication times. We also show that the algorithm exhibits almost linear scalability for large size graphs that can not be accommodated in a single machine. We also show that the use of GPUs results in upto 23% improvement in performance over multi-node CPU-only performance.

Following are the primary contributions of our work.

(1) We propose a novel divide-and-conquer approach for large-scale graph explorations using multiple nodes in which BSP has almost become a de-facto approach.
(2) Our algorithm involves a novel hierarchical merging that avoids the single-node space complexity bottlenecks of the usual merging strategies.
(3) The divide-and-conquer strategy, implemented using our HyPar framework involving simple API and efficient runtime strategies, can simultaneously harness multiple devices on multiple nodes.
(4) With these mentioned capabilities of our approach, our multi-node multi-device MST algorithm gives large-scale improvements over an existing BSP approach, exhibits good scalability, and demonstrates the performance benefits in utilizing both the CPU and GPU devices.

Section 2 gives related work in the area of parallel MST on different architectures. Section 3 describes our multi-node multi-device MST algorithm including the different steps of partitioning, independent computations and merging, and single node optimizations. Section 4 gives implementation details including our HyPar API and runtime strategies. Section 5 explains our experiments and results, including comparisons with an existing BSP approach, scalability analysis, and performance of our multi-node CPU-GPU executions. Section 6 gives conclusions and our future plans.

2 RELATED WORK

Parallel Boruvka's Minimum Spanning Tree in shared memory multi-core architecture was presented in [3, 6]. Galois[16] is a system for multi-core environments that incorporates the concept of operator formulation model in which an algorithm is expressed in terms of its action (or operator) on data structures.

One of the earliest works of Boruvka's MST on GPUs was by Vineet et al.[18]. An improved version was presented in Lonestar-GPU framework[14] using data-driven approach in which the GPU threads only explore the active vertices. The work by Sousa et al. [7] has shown performance improvements over Lonestar-GPU using effective graph contraction technique. The work by Pai et al.[15] presented optimized GPU kernels with different execution policies on GPU. Gunrock[19] is a single node multi-GPU framework which supports effective switching between vertex and edge frontiers.

Pregel[12] is an open source framework that follows BSP method to communicate among processors in a distributed memory environment. It organizes the computation into supersteps and communication is done after each superstep. GPS[17] is a popular framework which has explored MST application using a modified version of BSP approach. It uses a strategy called LALP (Large Adjacency List Partitioning) in which the high degree vertices are partitioned across machines. It also uses dynamic re-partitioning as the superstep progresses to balance the workloads. Pregel+[20] is another distributed framework which has explored minimum spanning forest application and has shown to outperform GPS. This framework also uses a modified version of BSP approach including strategies such as vertex mirroring and message combining with request response API to facilitate message pulling from remote vertices. It has also shown to outperform other existing popular distributed frameworks including Giraph[1] and PowerGraph[10] which have also tried to improve upon BSP approach. These approaches involve large communication and synchronization overheads.

3 MULTI-DEVICE MST ALGORITHM

The minimum spanning tree (MST) of a connected graph is the subset of edges that connects all the vertices such that the total sum of the weights of the edges of the subset is minimum among all such possible subsets. If the graph is disconnected then it finds minimum spanning tree within each connected parts and the collection of trees form minimum spanning forest (MSF). Given a graph G with V number of vertices and E number of weighted edges, the MST algorithm finds a subset V' which contains $V - 1$ edges (if the graph is connected) or $V - k$ edges (if the graph has k connected components).

Our multi-device algorithm is based on Boruvka's MST algorithm. 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 that form the end points of the lightest edge are then merged to form larger components. This operation is called *edge contraction*. 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).

Our multi-device algorithm consists of several steps including partitioning, independent computations, and hierarchical merging. The following sections describe these steps.

3.1 Partitioning

We have used the method explored by Gemini framework [21] for reading the graph in parallel by MPI processes. At first, each MPI process calculates the read offset using the rank of the process and reads from the input file. Then, the processes calculate the degrees of all the vertices globally using allreduce operation. Based on the degrees, a 1D partitioning scheme is used to balance the number of edges across computing units. As shown in [21], many large-scale real world networks posses natural locality where adjacent vertices are likely to be stored close to each other. Hence contiguous 1D partitioning effectively preserves the locality of these networks.

Within a node, we divide the graph into two partitions for the CPU and GPU based on the proportional performance of the given application for the given graph for the two devices. We once again employ 1D block partitioning in which we divide the CSR (Compressed Sparse Row) arrays representing the graph into two contiguous segments based on the ratio of CPU and GPU performance for the application.

After partitioning, each processor maintains information on the ghost edges. A ghost edge connects a boundary vertex of a partition to a vertex of another partition, called the ghost vertex. Each processor maintains the information on the ghost edges in a hash table indexed on the processor id of the ghost vertex. This hash table is called *ghostList*. For a processor to build its hash table, it has to receive the ghost vertices from the other processors. Since the number of boundary vertices can be large, the processors communicate these boundary vertices in multiple phases. After receiving the ghost vertices from the other processors, a processor parallely updates the ghostList using multiple threads that simultaneously process different boundary vertices.

3.2 Independent Computations on Multiple Devices

After partitioning the graph, we perform independent computations on the multiple devices of different nodes by invoking the Boruvka's MST algorithm independently on each device with the partition assigned to the device as the input graph. While performing independent Boruvka's MST algorithm on multiple devices, care must be taken to ensure that such independent computations do not result in incorrect results. For example, considering a partition of the graph as a subgraph and passing this subgraph as graph input to a Boruvka's MST implementation will result in incorrect results since the lightest edge from a component in a partition can connect to a vertex in another partition, i.e., can be a cut edge. We formulate a novel multi-device Boruvka's MST algorithm in which we add an exception condition 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, we stop the component from further expanding and proceed with the other components. Hence, at the end of the independent Boruvka's MST in a partition on a device, multiple components are obtained as output.

An example input graph is shown in Figure 1. Figure 2 shows the partitions formed on four devices, and components formed in each partition due to independent computations. We find that for partition 2, the the lightest edge for vertex 7 is to a component in partition 1. Hence, its component is not expanded further.

3.3 Merging

After the independent computations, the results formed on the different devices will have to be merged. This consists of two parts: 1. reducing the data structures to represent the smaller number of components, and 2. merging of the components of the different devices.

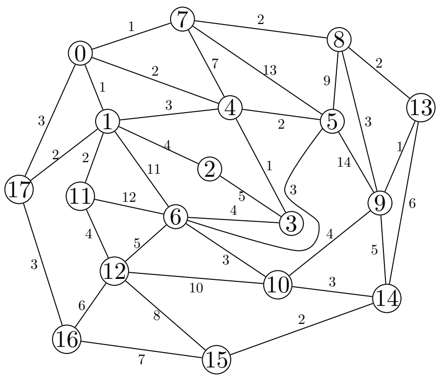


Figure 1: An Example Graph

Reducing the data structures consists of removal of self-edges and multiple edges. For subsequent stages, only inter-component edges are needed. The intra-component edges, called *self edges*, are removed in parallel by the multiple devices. Similarly, for a pair of components, only the lightest edge between the component needs to be retained. The other edges between the vertices of the two components can be removed. We denote this as *multi-edge removal*. For multi-edge removal, each component chooses a representative vertex as a parent vertex. For identifying multi-edges between two components residing in two different processors, the information related to the components of the ghost vertices of the processors must be communicated to each other. This information is communicated in the form of the parent ids. Thus, each of the two processors first obtains the corresponding ghost edges by indexing on the processor id of the other processor in its ghostList hash table, updates the parent information or component index of its boundary vertices in the ghost edges, and sends this information to the other processor. The receiving processor then uses the parent ids of the ghost vertices, retains only the lightest edge between the two components and removes the other edges. The communication of the ghost vertices happen in multiple phases due to the potentially large number of boundary vertices. We make use of another hash table to maintain minimum weight edge between a pair of components. We explore the vertices in parallel, obtain their adjacency list and update the hash-table with the minimum weighted edges to the other components.

3.4 Hierarchical Merging of Components in Different Processors

The components formed in the different devices will have to be merged to form larger components. One way is to make all the devices communicate their components to a single node and perform independent computations by invoking Borouvka's MST on all the components in the single node. However, this strategy has two drawbacks: 1. in many cases, a single node will not be able to accommodate all the components formed in all the devices, and 2. merging in a single node results in under-utilization of the other devices.

We propose a novel hierarchical strategy in which we first form groups of active processors. Initially, all the processors are denoted

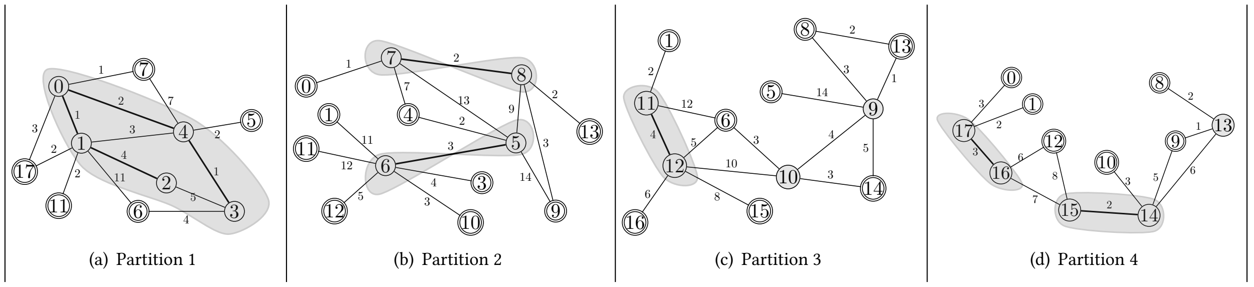

(a) Partition 1 (b) Partition 2 (c) Partition 3 (d) Partition 4

Figure 2: Partitions and independent computations for four Devices. The shaded regions denote the components. Ghost vertices for a partition are represented as unshaded double-line vertices.

as active. We experimented with different group sizes of 2, 4, 8 and 16, and chose a group size of 4 based on average performance. In each group, the processors exchange some of their components and perform collaborative merging of the components by performing independent computations. The method is inspired by Rabenseifner's algorithm for reduce and allreduce [2]. The fundamental concept in Rabenseifner's algorithm is to divide the input array in each processor into segments, exchange the segments between the processors, and perform simultaneous reduction operations with respect to different segments on different processors. For example, considering two processors, P_0 and P_1, with input arrays A and B, respectively, processor P_0 divides its array into two equal segments, $A1$ and $A2$, and P_2 similarly forms two segments $B1$ and $B2$. P_0 then sends $A2$ to P_1, and P_1 sends B_1 to P_0. The processors then parallely perform reduction operations with respect to the two segments: P_0 performs reductions with A_1 and B_1, and P_1 performs reduction operations with A_2 and B_2.

For MST, the processors in a group divides its components into segments and exchange the segments. The segments are formed such that a processor will be able to accommodate at least one segment it receives from another processor in addition to the segments that it contains. Unlike the reduction operations, the segments in one processor can have dependencies with multiple segments of another processor, since a component can have connections to multiple components. Hence, we follow multiple rounds of exchanges between the processors.

We follow a ring-based communication, in which a processor, P_i, receives a segment of components from its right neighbor, $P_{(i+1) mod P}$, and sends one of its segments to its left neighbor, $P_{(i-1) mod P}$. The processors then simultaneously perform independent computations by invoking Boruvka's MST with its new set of components consisting of its own and the received set of components. We continue this ring-based communication and collaborative merging until all the components in a group can be accommodated in a single node, designated as the group leader, and based on a threshold, as described in Section 4.3.4. At this stage, the processors communicate their components to their group leader, which then performs independent computations using Boruvka's MST on all the components.

In the next stage, the leaders are designated as active processors, and groups are formed out of the leader processors. We then continue the above procedure of group based collaborative merging using these newly formed groups. This process continues until only one active processor remains at which stage the remaining components are moved to this processor. The single processor then performs independent computations using Boruvka's MST algorithm with the remaining components to form the final result.

The working of the hierarchical merging is illustrated in Figure 3. Level 1 in the figure shows the reduced graph after independent computations followed by self edge and multi edge removal of the partitioned graphs shown in Figure 2. We explain the ring-based collaborative merging with a group size of 2 in Figure 3. For partition 2, components 5 and 7 are divided into segments and one of the segments corresponding to component 5 is exchanged with partition 1. Similarly, partition 3 exchanges its segment corresponding to component 11 with partition 4 and receives component 13 from partition 4. Level 2 shows the updated graph after the exchange. Then each of the partition performs independent computations, self edge and multi edge removal. After these steps, the components in a group are merged to the leader of the group. Level 3 shows the updated data after merging. We again perform independent computation steps on the leader nodes and follow the same strategy for exchanging segments and collaborative merging but in a different group with only the leader nodes. Level 4 shows the updated data after the exchange in leader nodes. We follow the same strategy up to the level of hierarchy until the data is merged to a single node. Level 5 shows the final level of the hierarchy containing the final merged data in a single partition. Boruvka's MST kernel is invoked on this partition to compute the final set of lightest edges among the remaining components.

3.5 Single Node CPU-GPU Executions

For executions within a single node, the above steps are carried within the node where the partition assigned to the node is further divided into CPU and GPU partitions. Both the devices then perform independent computations using Boruvka's MST algorithm and self-edge removal. They also communicate ghost edges and perform multi-edge removal. The components are then merged in one of the devices and post processing is performed as described in section 4.1. The CPU algorithm is based on Galois' implementation [16]. The GPU algorithm uses the worklist based approach with generic optimizations. We performed different optimizations in implementation of the graph kernel functions, primarily related to GPU kernels.

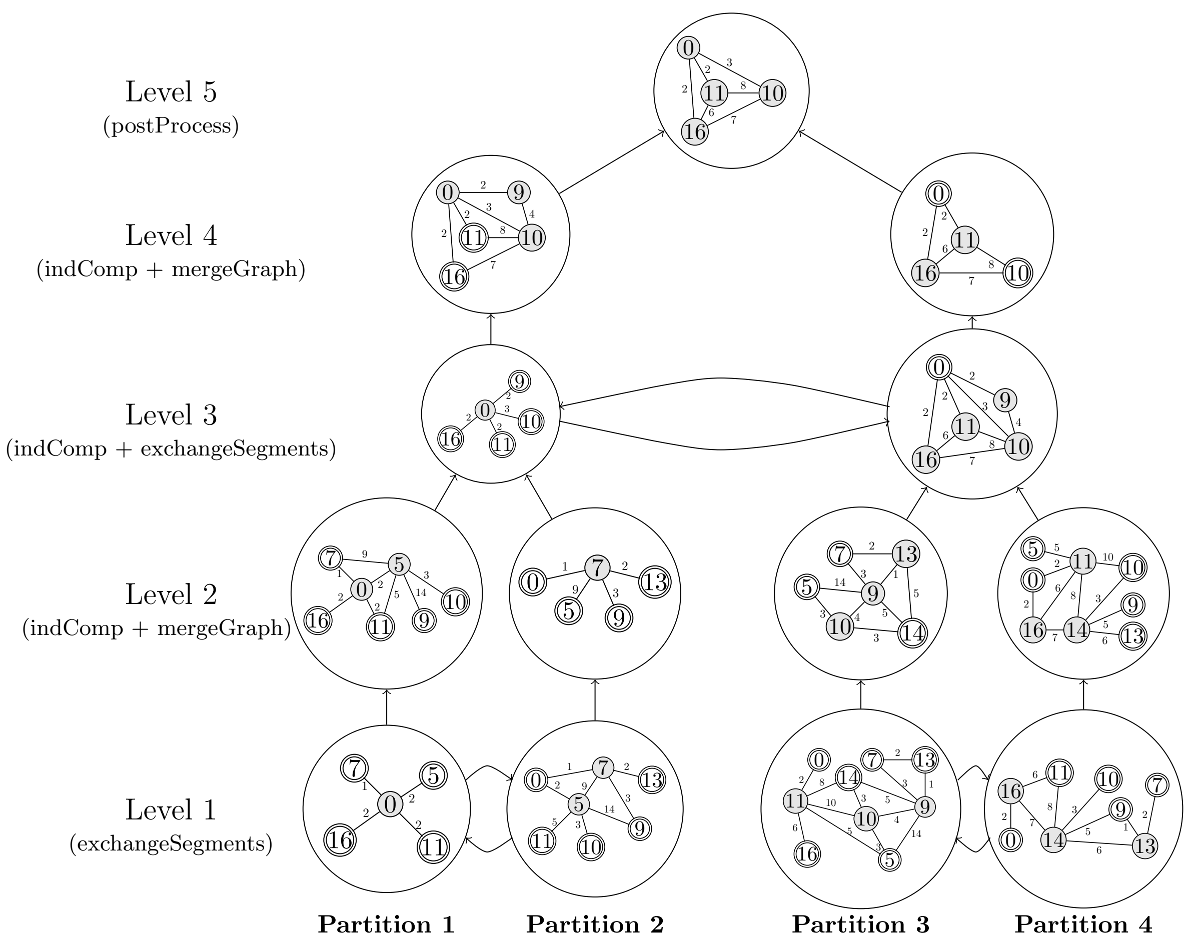


Figure 3: Illustration of hierarchical merging. Ghost vertices for a partition are represented as unshaded double-line vertices.

Hierarchical Strategy for Processing Adjacency List: For large real-world graphs which often has varying degree distribution, exploring adjacency of every vertex by a single thread can result in under-utilization of GPU resources. We employ a hierarchical strategy to assign different number of threads to explore adjacency of a vertex depending on its degree [13].

Data-driven and Worklist Approach: The GPU computations can be organized as *topology-driven* or *data-driven* computations. In the topology-driven algorithms [11], GPU threads are spawned for all nodes in a graph, while in the data-driven algorithms[14], worklists are built dynamically and threads are spawned corresponding to only active elements/nodes that have to be processed in a time step. We use the data-driven approach 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, we minimized the number of atomic accesses by batching atomic accesses into a single atomic access and performing hierarchical atomic accesses [9].

We also minimized the time CPU-GPU data transfers by identifying different parts of the independent computations in the GPU and the different data needed by the computations, and overlapping parts of the computations with the transfers of data that are not needed by the computations using cudaStream.

CPU Optimizations: Galois [16] is a programming approach that uses amorphous data-parallelism that involves defining active elements, and applying operators on the active elements. It uses the data-driven approach of maintaining a worklist of active nodes. For our CPU based implementation of all our graph kernels, we have used the Galois data-driven and unordered worklist based approach using OpenMP threads.

4 IMPLEMENTATION

Our distributed multi-device MST algorithm has been implemented on our framework, called *HyPar*, a programming and runtime framework for hybrid CPU-GPU executions of graph applications. The framework provides a simple and generic API containing a small number of functions. The framework is supported with efficient hybrid runtime strategies including 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.

4.1 HyPar API

HyPar provides four functions corresponding to the steps of multi-device graph processing. The API functions are shown in Table 1.

4.1.1 Partitioning the Graph. The *partGraph* function divides the graph into partitions for the devices using 1D block partitioning and based on the proportional performance of the given application for the given graph for the devices.

4.1.2 Independent Computations. After partitioning the graph, our framework sends the respective partitions to the devices. The *indComp* function then executes the application independently on the devices without any communication between the devices. Within each node, we assign one of the CPU threads, denoted as *GPUdriverThread*, to drive the GPU execution, and the other CPU threads, denoted as *processingThreads*, for executing the CPU multi-core version.

The *indComp* function also has an optional *excpCond* argument. Note that the independent computation on a device involves execution of a graph application/algorithm like BFS, MST etc. on a partition assigned to the device. However, execution of the original graph algorithm as such while treating the partition 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 partition 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 partition. Our framework also provides $EXCPT\_BORDER\_EDGE$ exception condition.

4.1.3 Merge. The *mergeParts* function merges the results obtained on the devices due to the independent computations. In each node, our framework copies the required information from the GPU device to the host and merges to an internal data structure. After merging, we update the graph data structure in the CPU using optimized parallel graph update functions implemented in our framework. In MST, we contract each component to a single vertex and remove all the internal edges of the community using parallel thread operations. We also remove multiple edges across the communities.

4.1.4 Post Processing. After the merge step, if the remaining graph data size is reasonably small the framework runs the algorithm given by the $postProcessKernelName$ on one of the devices using the remaining data. The final output is made available in the CPU in the $finalOutput$ argument.

4.2 Multi-Node Multi-Device MST with HyPar

Algorithm 1 shows the multi-node multi-device MST (MND-MST) algorithm using HyPar API.

Algorithm 1 MND-MST algorithm using HyPar

```
procedure MERGEPARTS(G, current_set, cpuRep, rank, nProcessors, ghostList)
    removeSelfEdges(G, current_set, cpuRep)
    createGhostMessage(G,ghostList,cpuRep,rank)
    sendGhostMessage(G, ghostList, cpuRep, rank, nProcessors)
    removeMultiEdges(G, current_set, cpuRep)
    gEdges ← calculate G.edges in a group
    if $gEdges > threshold$ then
        exchangeSegmentsWithinGroups(G, current_set, cpuRep, rank, nProcessors)
    else
        return mergeGroup = true
    end if
end procedure

procedure MST(G)
    MPI_Instance newMPI(argc,argv)
    G=readGraph(inputFile, nVertices, rank, nProcessors);
    partRatio ← partGraph(G)
    initRep(G,cpuRep)                                   ▷ Initialize with vertex id
    repeat
        Initialize ghostList data structure
        makeGhostInformation(G,current_set,ghostList,rank, nProcessors)
        indComp(G,current_set,cpuRep,EXCPT_BORDER_VERTEX)
        mergeParts(G,current_set, cpuRep, rank, nProcessors, ghostList)
        if $mergeGroup$ then
            mergeGraph(G,current_set, cpuRep, rank, nProcessors, groupSize)
        end if
    until data is gathered to a single machine
    postProcess(G,current_set,cpuRep,EXCPT_BORDER_VERTEX, finalOutput)
end procedure
```

4.3 HyPar Runtime

HyPar follows several runtime strategies for realizing the APIs for efficient multi-device executions.

4.3.1 Ratio for Graph Partitioning. We assume a homogeneous set of nodes in which the graph is partitioned equally among the nodes. Within a node, to determine the ratio of CPU-GPU performance, we form a small number of different induced subgraphs (for our study, we used 5-10 subgraphs), execute each subgraph on both CPU and GPU, find the performance ratio, and obtain an average of the ratios for the subgraphs. Each subgraph is generated randomly such that the number of vertices in the subgraph is 5% of the total number of vertices in the original graph. In addition to performance, we also take into account the GPU memory requirements for the problem.

4.3.2 Threshold for Independent Computations. The independent computations are performed on the devices 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 decreases 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 performing the merging step.

4.3.3 Recursive Invocation of Partitioning - Independent Computations - Merging. After merging within a node, we update the graph data structure and keep only the required vertices and its outgoing edges. Using experiments we found that if the reduced

Function	Remarks
$partGraph(graph)$	Partitions the graph into multiple partitions for multiple nodes, and within each node creates two partitions for the CPU and the GPU.
$indComp(graph, current\_set, indCompResult, excpCond)$	Performs independent computations of a graph kernel on the partitions. Returns the result in $indCompResult$.
$mergeParts(graph, current\_set, indCompResult, rank, nProcessors, ghostList)$	Merges the results from the independent computations on the devices and communicates ghost vertices.
$postProcess(graph, current\_set, indCompResult, excpCond, finalOutput)$	Performs post-processing by executing the kernel on the vertices in $current\_set$ and the final result is available in $finalOutput$.

Table 1: HyPar-API Functions

graph after the merge step is sufficiently large, it is beneficial to invoke independent computations again using the reduced graph. Our HyPar-runtime follows this recursive approach by again partitioning the reduced graph using already calculated partitioning ratio and followed by *indComp* and *mergeParts* steps. For our current work, we use the number of edges in the reduced graph (specifically, a threshold of 100 million edges) to decide to continue with recursion or to move to the post processing step.

4.3.4 Threshold For Hierarchical Merging. Our multi-node algorithm has a phase of exchanging segments between the processors in a group using a ring-based algorithm, and performing independent computations and collaborative merging with the new set of segments. It then switches to a phase of moving all the segments in the group to the group leader and performing independent computations. Our HyPar runtime automatically switches from the former to the latter phase if the size of the data in the group is less than a threshold. To find the threshold we use a simple strategy based on a convergence criteria. If after exchange of segments and collaborative merging, the size of the data does not reduce significantly, the exchanges between the processors of the group are stopped and the data is merged to the leader.

5 EXPERIMENTS AND RESULTS

5.1 Experimental Setup

We have uses two platforms for our experiments. We have used a 16 node cluster consisting of AMD Opteron(tm) 3380 processor. Each node is equipped with 8 CPU cores operating at 2.6GHz and 32GB of Main Memory. We have also used a CrayXC40 supercomputing system to evaluate our multi-device code. We use 16 nodes and each node is equipped with one Intel Xeon Ivybridge E5-2695 v2 processor and one Nvidia Tesla K40 GPU accelerator card. The CPU processor has 12 cores running at 2.4GHz with 64GB main memory. The accelerator card has 2880 cores with 12GB Device Memory.

The graphs used in our experiments are shown in Table 2. The graphs were obtained from the University of Florida Sparse Matrix Collection [8] and the Laboratory for Web Algorithmics [5][4]. As shown in the table, we have used several real world graphs from different categories and having different characteristics including varying degrees for our experiments. All the graphs are large-sized graphs with mostly billions of edges that cannot fit within a single node. We have converted these graphs to undirected graphs and assigned random weights to the edges. All the results shown are obtained using averages of three runs.

Graph	$\|V\|$	$\|E\|$	Approx. Diam.	Avg. Deg.	Max. Deg.
road_usa	23.9M	57.7M	6262	2.41	9
gsh-2015-tpd	30.8M	1.16B	9	37.73	2176721
arabic-2005	22.7M	1.26B	29	55.50	575662
it-2004	41.2M	2.27B	27	55.01	1326756
sk-2005	50.6M	3.62B	17.56	71.49	8563816
uk-2007	105M	6.60B	22.78	62.76	975419

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

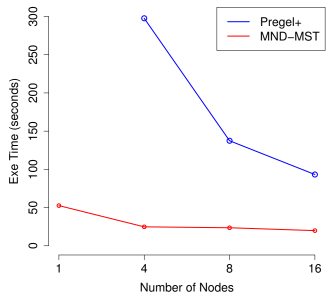
Graph	Pregel+ Exe Time	Pregel+ Comm Time	MND-MST Exe Time	MND-MST Comm Time
road_usa	113.19	76.82	21.56	8.07
gsh-2015-tpd	112.53	79.09	84.49	47.29
arabic-2005	93.26	67.95	19.83	9.52
it-2004	161.09	113.99	40.20	15.95
sk-2005	272.04	207.49	45.78	17.96
uk-2007	523.63	321.73	60.39	24.53

Table 3: Performance Comparison with Pregel+

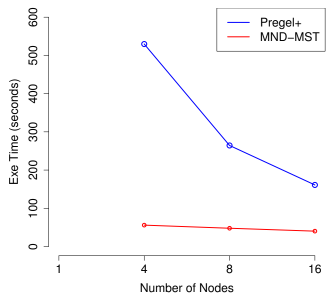
5.2 Comparison with Pregel+ [20]:

To compare with Pregel+, we have run our experiments on the AMD cluster. We have run Pregel+ on 16 nodes with 8 workers on each node and one of the machines acted as the master while the other nodes acted as slaves. We compiled Pregel+ with Hadoop-2.6.1 and gcc 4.8.5. We ran our multi-node CPU only version of MND-MST with 8 OpenMP threads on each node. Table 3 shows the comparison results with Pregel+.

As shown in Table 3, our MND-MST version achieves a performance improvement of 75-88% over Pregel+ except for gsh-2015 graph. For gsh-2015 graph we achieve 24% performance improvement. As shown in the table, we get these large scale benefits over Pregel+ due to the reduction of communication time. Pregel+ follows BSP approach and has large communication and synchronization overheads. Our MND-MST algorithm improves the communication time over Pregel+ by 85-92% with an average of 89% except for the gsh-2015 graph. For the gsh-2015 graph, we obtain about 40% reduction in communication time. We get relatively less performance improvement for gsh-2015 because the independent computations do not yield large components for this graph. Hence, it requires more number of exchange of segments and collaborative merging in a group at initial iterations thereby increasing communication time. For the other graphs, the MND-MST algorithm



(a) arabic-2005

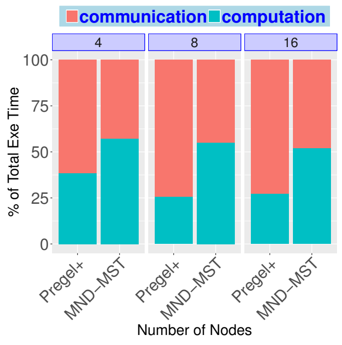


(b) it-2004

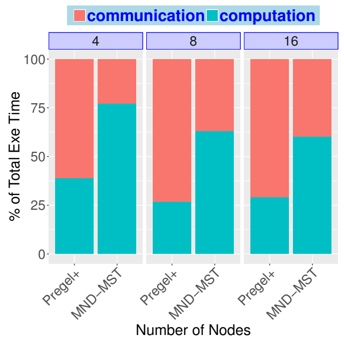
Figure 4: Inter-node Scalability of Pregel+ and MND-MST

Nodes	arabic-2005	it-2004
1	52.60	
4	24.82	55.92
8	23.62	47.80
16	19.88	40.20

Table 4: Performance of MND-MST with increasing number of Nodes



(a) arabic-2005



(b) it-2004

Figure 5: Computation vs Communication for Pregel+ and MND-MST

forms larger components in *indComp* routine and achieves larger performance benefits over the prevalent BSP approach.

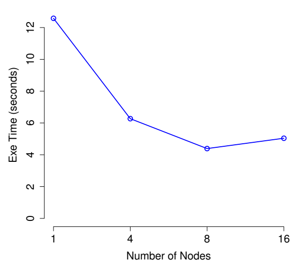
Figure 4 shows the scalability comparison of MND-MST with Pregel+ upto 16 nodes. Pregel+ obtains good scalability from 4 to 8 nodes for both the graphs shown in the figure. But the scalability improvement comes with added overheads. Due to these overheads, we are not able to run Pregel+ for arabic-2005 graph in a single node, as shown in Figure 4(a). Our single node MND-MST version completes faster than Pregel+ on 16 nodes. Table 4 shows the total execution time of MND-MST with increasing number of nodes in the AMD cluster. As shown in the table, for arabic-2005 graph we obtain 2.12x performance improvement on 4 nodes and 2.64x performance improvement on 16 nodes compared to our single node version.

In BSP approaches, the computing processes need to communicate after every superstep. Thus BSP approaches can incur large communication and synchronization overheads. As shown in Figure 5, for both of these graphs, Pregel+ takes more time for inter-node communications rather than doing useful computations. For 16 nodes, it spends about 75% of the total execution time for communications, and only 25-32% in useful computations. We obtain large scale benefits over Pregel+, as in our novel divide-and-conquer strategy for multi-node execution, the processors spend 62-75% of the execution time in performing useful computations.

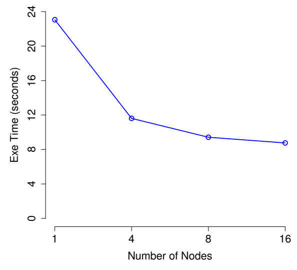
5.3 Scalability for CPU-only MND-MST

In this section, we analyze the scalability of MND-MST CPU-only version on the Cray cluster. Figure 6 shows the scalability for the large graphs. As shown in the figure, we obtain good scalability for these graphs. We could not accommodate the last two graphs in a single node of the Cray cluster. For sk-2005 graph, we achieve speed up of 1.31 and 1.9 for 8 and 16 nodes, respectively, compared to 4 nodes. The scalability is even larger for the uk-2007 graphs with speedups of of 1.54 and 2.11 at 8 and 16 nodes, respectively, compared to 4 nodes.

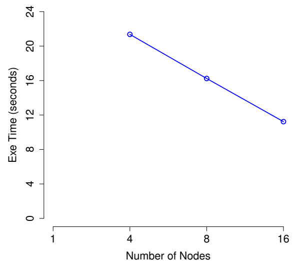
For road_usa graph, we obtain slowdowns for larger number of nodes. As the size of the graph is very small, with increasing number of nodes ,it requires very less time for computations and takes more time in communications. This is illustrated in Figure 7. For this graph, with more number of partitions, the *indComp* kernel has less work and most of the work is done by *postProcess* kernel. The MND-MST algorithm is not able to form larger components and the algorithm has to rely on postProcessor kernel



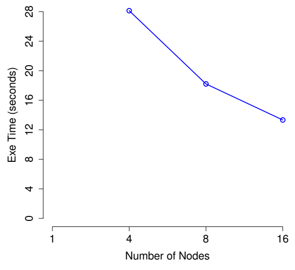
(a) arabic-2005



(b) it-2004



(c) sk-2005



(d) uk-2007

Figure 6: Scalability of CPU only MND-MST on Cray

when the reduced data is merged to a single node. For gsh-2015 graph, we obtain slow down on 4 nodes compared to its single node performance. But, we obtain speedups with further increase in the number of computing nodes. This is explained in Figure 7. For gsh-2015 the *indComp* kernel forms many smaller components on each node. Hence, it requires more number of iterations for hierarchical merging. As hierarchical merging requires exchanges of segments with computing nodes, it also takes significant time for communication as shown in the figure. With increasing number of nodes, the work for *indComp* kernel even decreases. But as the data structures are reduced after each *indComp* routine, the size of the data reduces at lower levels of the hierarchy with the help of more computing nodes. Hence, it takes lesser time for communication with increasing number of nodes. As smaller components are formed, the work for *postProcess* kernel increase with increasing number of nodes. For the larger size graphs, most of the execution time is for the *indComp* kernel. As shown in Figure 7, for uk-2007 graph, the processors are mostly involved in performing *indComp*. As large size components are formed in each partition, very less times are taken for for communications and the *postProcess* kernel, thereby achieving good scalability.

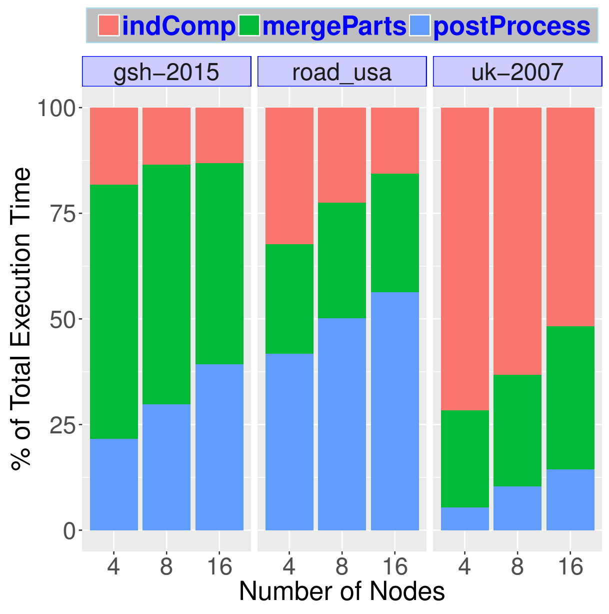


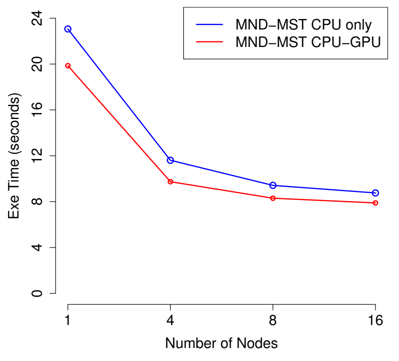
Figure 7: Execution Time for Different Phases

5.4 Scalability for Multi-device(CPU-GPU) Execution

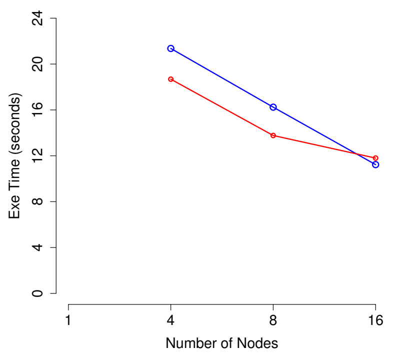
Next, we analyze the scalability of MND-MST CPU-GPU version and compare with MND-MST CPU only version. As shown in Figure 8, we find good scalability with our CPU-GPU version. We utilize the GPUs only for *indComp* and possibly for *postProcess* kernel in the distributed environment as explained in Section 4.1. As shown in Figure 8(a), our CPU-GPU version gives 14% performance improvement over the our CPU-only version for the it-2004 graph in a single node. With the increase in the number of nodes, the amount of computations by the *indComp* kernel decreases. Hence, the performance improvement is only 10% over our CPU-only version for 16 nodes. For large size graphs, as the *indComp* kernel requires more time we obtain larger benefits by involving GPU. This is shown in Figure 8(c), where we obtain we obtain 15.5% performance improvement on 4 nodes for the uk-2007 graph. The scalability of it-2004 and uk-2007 graphs follows the same pattern. For sk-2005 graph we obtain 15% performance improvement over CPU-only version for upto 8 nodes, but with 16 nodes both the CPU-only and CPU-GPU versions perform equivalently. Overall, with our CPU-GPU version of MND-MST, we obtain performance improvements upto 23% with an average of 9% over the CPU-only version for all the graphs in Table 2.

6 CONCLUSIONS AND FUTURE WORK

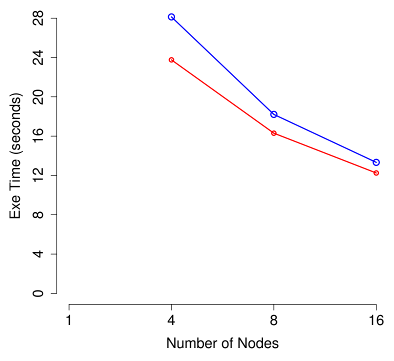
In this paper, we presented a novel strategy for Minimum Spanning Tree algorithm on multi-node multi-device settings, with divide-and-conquer approach. Our experiments show that our novel strategy provides average performance improvements of 71% over the state-of-art, best performing distributed approach which follows Bulk Synchronous Processing(BSP) method for inter-node communications. Our method improves over the communication and synchronization cost involved with BSP using the strategy of independent processing and obtains good scalability. When extended



(a) it-2004



(b) sk-2005



(c) uk-2007

Figure 8: Scalability Comparison of MND-MST CPU only and MND-MST CPU-GPU versions on Cray

to multiple devices in distributed settings, we obtain performance improvements upto 23% over our CPU-only version. We plan to extend this work to to implement more graph applications in future. We also plan to develop APIs and runtimes for graphs applications that are not amenable for divide-and-conquer executions.

REFERENCES

[1] [n. d.]. Giraph Homepage. http://giraph.apache.org/. [Online].

[2] [n. d.]. Rolf Rabenseifner. New optimized MPI reduce algorithm. https://fs.hlrs.de/projects/par/mpi/myreduce.html. [Online].

[3] David A Bader and Guojing Cong. 2006. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *J. Parallel and Distrib. Comput.* 66, 11 (2006), 1366–1378.

[4] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). 587–596.

[5] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. 595–601.

[6] Guojing Cong and David Bader. 2004. Lock-free parallel algorithms: An experimental study. In *International Conference on High-Performance Computing*. Springer, 516–527.

[7] Cristiano da Silva Sousa, Artur Mariano, and Alberto Proença. 2015. A Generic and Highly Efficient Parallel Variant of Boruvka's Algorithm. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*. IEEE, 610–617.

[8] Timothy A Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1.

[9] Ian J. Egielski, Jesse Huang, and Eddy Z. Zhang. 2014. Massive Atomics for Massive Parallelism on GPUs. (2014), 93–103.

[10] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: distributed graph-parallel computation on natural graphs.. In *OSDI*, Vol. 12. 2.

[11] Pawan Harish and PJ Narayanan. 2007. Accelerating Large Graph Algorithms on the GPU using CUDA. In *International Conference on High-Performance Computing*. 197–208.

[12] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[13] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU Graph Traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. 117–128.

[14] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Data-driven versus Topology-driven Irregular Computations on GPUs. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. 463–474.

[15] Sreepathi Pai and Keshav Pingali. 2016. A compiler for throughput optimization of graph algorithms on GPUs. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 1–19.

[16] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. Amber Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. 2011. 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*. 12–25.

[17] Semih Salihoglu and Jennifer Widom. 2013. GPS: a graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM, 22.

[18] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and PJ Narayanan. 2009. Fast minimum spanning tree for large graphs on the GPU. In *Proceedings of the Conference on High Performance Graphics 2009*. ACM, 167–171.

[19] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. 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*. 11.

[20] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2015. Effective techniques for message reduction and load balancing in distributed graph computation. In *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 1307–1317.

[21] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System.. In *OSDI*. 301–316.