Efficient Multi-GPU Graph Processing with Remote Work Stealing

Ke Meng, Liang Geng, Xue Li, Qian Tao, Wenyuan Yu, Jingren Zhou
Alibaba Group
{mengke.mk, guanyi.gl, youli.lx, qian.tao, wenyuan.ywy, jingren.zhou}@alibaba-inc.com

2023 IEEE 39th International Conference on Data Engineering (ICDE) | 979-8-3503-2227-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICDE55515.2023.00022

***Abstract*—Graph algorithms support a broad spectrum of big data applications. A typical approach to scale graph algorithms is to run in a distributed and parallel setting with multiple processing devices. The approach requires balanced and effective utilization of computation, memory, and communication resources across devices. To address the problem, a large number of studies have been conducted, such as graph partitioning and asynchronous computation. However, there are still many outstanding issues yet to be solved. For example, the workloads can be skewed differently across devices, and between iterations, even with the state-of-the-art graph partitioners. As the graph partitions are typically static, they fall short in capturing the dynamic characteristics with different algorithms, inputs, and progress, leading to poor utilization of resources. Recently, GPUs have been increasingly used to accelerate various graph algorithms. Their highly efficient interconnection technologies, such as NVLink, open new opportunities for us to achieve better resource utilization. In this paper, we analyze the dynamic load-imbalance (DLB) problem and the long tail (LT) problem in multi-GPUs and solve them by adaptive *remote work stealing* on-the-fly. We first introduce a frontier stealing algorithm to solve the DLB problem, then an ownership stealing algorithm to solve the LT problem. Based on these two algorithms, we developed GUM — a multi-GPU graph processing system with high device utilization. We evaluated GUM on four typical graph algorithms (BFS, WCC, PR, SSSP). The results show that GUM can run up to an order of magnitude faster than Gunrock and Groute, with fewer stragglers and less synchronization overhead.**

***Index Terms*—Graph processing, Multi-GPU, Load balance**

I. INTRODUCTION

Graph algorithms serve as essential building blocks for a wide range of applications, such as social network analytics [1], routing [2], constructing protein network and De Bruijn graphs [3], and mining valuable information in RDF (Resource Description Framework) graphs [4]. Generally, graph analytics involve propagating labels across edges or iteratively accumulating values from adjacent vertices. However, in the era of big data, the computational and storage complexity of sophisticated algorithms coupled with rapidly growing datasets have exhausted the limits of a single device.

To process larger graphs, distributed graph processing has been employed by recent graph processing systems *e.g.,* Gemini [5], and Gunrock [6], [7]. That is, we partition a large graph into multiple subgraphs, then process each subgraph in one computing device (*e.g.,* CPU, GPU) in a data-parallel manner. However, the performance of distributed graph processing is often frustrating mainly due to the low utilization of computing devices. Specifically, in distributed graph computing, computing devices lose efficiency mainly due to waiting for *e.g.,* synchronization, and stragglers. For example, messages in distributed graph processing often need to be serialized before communication, which requires manipulating many message buffers, resulting in considerable overhead, thus computing devices have to wait for the completion of such data movement before advancing to the next round. A recent work [5] shows that systems lose nearly a third of device utilization, as they move from a single node to distributed implementations.

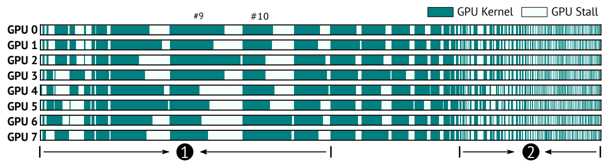

Fig. 1: The timeline of SSSP of Gunrock.

Example 1: Figure 1 shows the timeline of the single-source shortest path (SSSP) algorithm of Gunrock [6], [7] on the *webbase* [8] graph. The input graph is well-partitioned with each GPU processing the same amount of edges. However, we can still observe that the computing devices stall due to:

The dynamic load-balance (DLB) problem (labeled as ❶). The load imbalance of different computing devices leads to stragglers. Recent works [4], [9], [10], [11], [7], [12], [13], [14], [5] rely heavily on static graph partitioners to solve the load imbalance problem. We find that the load-imbalance can still be severe even if the graph has been well-partitioned with advanced partitioners. In most cases, only a fraction of the vertices, generally called *frontiers*, are involved in the current iteration of graph computation. As shown in Figure 1, in the beginning, the frontiers start from a single source vertex. As it grows, the size of frontiers changes rapidly, causing imbalanced workloads, due to the irregularity of graph structure and the dynamic nature of graph algorithms [15]. For example, on iteration #9, the slowest worker (GPU_2) takes 2.6 $\times$ runtimes compared to the fastest worker (GPU_4). However, on iteration #10, the slowest worker (GPU_2) becomes the fastest worker and is 1.9 $\times$ faster than the slowest worker (GPU_6). Thus there is no guarantee that the frontiers are universally evenly distributed with a static graph partitioning algorithm. In our observation, the difference in the workload in different devices can be up to 4.2 times, which causes severe starvation.

The long tail (LT) problem (labeled as ❷). Graph traversal algorithms such as SSSP may take many iterations to converge. At the late stage of such graph algorithms, only a small fraction of vertices are activated. Instead of the computation, the latency gradually dominates the runtime, since the computational power is over abundance under this situation.

The latency mainly consists of synchronization overhead (*e.g.,* preparing message buffers for communication), or inevitable data movement (*e.g.,* transferring data between CPU and GPU). These overheads take about several milliseconds which is negligible in bustling iterations but is considerable when there are thousands of such latency-bound iterations. These synchronization overheads account for 21% of the total time in this example. Generally, the more devices involved in computation, the longer the latency is. The long-tailed phenomenon significantly limits the scalability of graph algorithms, such as delta-PageRank, SSSP, and BFS.

Both problems exist in CPU-based graph processing systems [14], [13], [5], [4], [16], [17], [18] and GPU-based graph processing systems [19], [20], [21], [6], [12], [22], [23], [24]. Moreover, these problems get even worse in GPUs, since GPU-based graph processing systems are more sensitive to load-imbalance and involve more overhead like synchronization between host and GPU. Fortunately, recent interconnection technologies for multi-GPU such as NVLinks [25] deliver high bandwidth and low latency, which provides new opportunities to make trade-offs between efficiency and communication.

GUM. To solve these problems, we develop GUM, a framework which leverages work stealing mechanism to solve the DLB and LT issues at the same time. Specifically, GUM seize the opportunities that: (1) ***The links between GPU pairs are asymmetric***, *e.g.,* there may be two lanes of NVLinks (50 GB/s) or a single lane of NVLink (25 GB/s) or no links between any two GPUs, as depicted in Figure 2. These connections vary significantly in speed, resulting in communication between some GPU pairs being more expensive than others. (2) ***Multiple stealing paths exist between a GPU pair.*** As shown in Figure 2, GPU_0 can steal some of the workload from GPU_7 by taking either GPU_1 or GPU_6 as intermediate transit. The performance of different stealing paths may vary significantly as the transmission data volume in each path may vary in each iteration.

Contributions & organization. This paper aims to solve the DLB problem and the LT problem. Compared with previous works, this study has the following contributions:

(1) The DLB problem (Section III). We proffer a frontier stealing (`FSteal`) strategy to solve the DLB problem. We first introduce a cost model of `FSteal` to characterize the computational and data-accessing pattern. Then, we formulate the optimization objective of `FSteal` and provide a linear programming approach to achieve it.

(2) The LT problem (Section IV). We proffer an ownership stealing (`OSteal`) strategy to solve the LT problem. Similarly, we introduce a cost model of `OSteal` and the optimization objectives. After that, we propose an algorithm with a greedy strategy to achieve it.

(3) Implementation (Section V). Based on `FSteal` and `OSteal`, we propose our multi-GPU graph processing system GUM. We describe the implementation details of GUM as well as several optimizations to improve our stealing mechanisms.

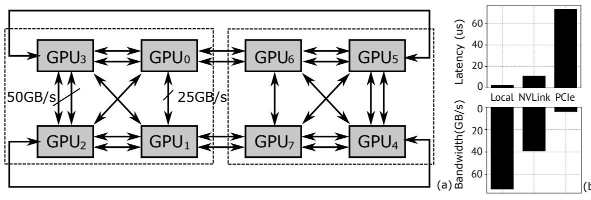

Fig. 2: An example of NVLink topology.

(4) Experimental study (Section VI). An extensive evaluation of the multi-GPU graph processing system GUM is presented, demonstrating its efficiency.

II. BACKGROUND

Graphs. A graph can be represented as $G = (V, E)$ where V represents a finite set of vertices and $E \subseteq V \times V$ is a set of edges. In real-life, $\{V, E\}$, is immutable and often very sparse, which means $|E| \ll |V| \times |V|$. In this work, we only consider the graphs that can fit the aggregated device memory.

Graph algorithms. A graph algorithm $\mathcal{A}$ is a set of procedures on G to do data analytics and solve real-world problems [26]. Since the GAS abstraction (*Gather-Apply-Scatter* [14]) of graph algorithm is widely used and achieves good results in real-world applications, this work considers running GAS graph algorithms on multiple GPUs in the BSP (Bulk Synchronous Parallel) [27] mode. More specifically, only a subset of V participates in the computation of an iteration k, which is called frontier, namely f_k, and $f_k \subseteq V$. `Gather` is to combine the incoming messages, on which an aggregation function is defined to aggregate the information collected by current frontiers through the edges. Such as picking out the minimum, or maximum value of messages passed by adjacent edges. `Apply` is to apply the aggregated results to the current frontiers, *i.e.,* update the distance of each vertex in an SSSP algorithm. `Scatter` propagates the latest result of the frontiers along all the edges of them.

Graph partitions. To process large graphs in a distributed system that has n workers, the input graph should be partitioned into n non-overlapping fragments (a.k.a. subgraphs). A n-way graph partition $(F_1, \cdots, F_n)$ of a graph $G(V, E)$ partitions G into n fragments, such that (a) $F_i = (V_i, E_i)$, (b) $V = \bigcup_{i=1}^{n} V_i$, and (c) $E = \bigcup_{i=1}^{n} E_i$.

There are two main schemes for graph partitioning, namely *edge-cut* and *vertex-cut*. In an edge-cut partition, the vertices of fragments are disjoint, such that each vertex only belongs to one fragment. And for a vertex-cut partition, the edges are disjoint, such that each edge only belongs to one fragment. A good graph partition is one in which the size of F_i is more evenly distributed, while the number of edges (vertices) replicated across fragments is as small as possible. In this work, we only consider the edge-cut partition scheme.

Graph partition techniques are widely used to solve load imbalance issues. However, graph partitioners often fail on the DLB problem and LT problem as shown in Example 1. The reasons behind the inapplicability are: (1) graph partitioners are static thus they can not adapt to the dynamic input such

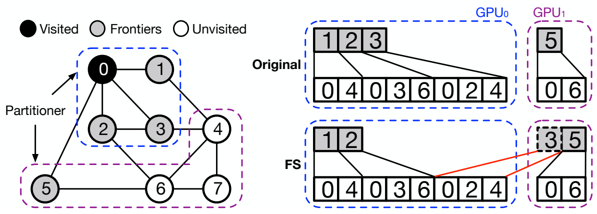

Fig. 3: An example stealing policy to balance the work.

as BFS/SSSP source, which may generate a completely different workload distribution. (2) graph partitioners usually try to preserve the locality of graph topology, making nearby vertices resident on the same partition, which may aggravate the "cocooning effect" (only one partition hold active vertices in the first few iterations). (3) Sophisticated graph partitioners often introduce high overhead [28]. Work Stealing is orthogonal to the graph partitioners, since it can work on either a good or bad graph partition scheme. But most importantly, work stealing is dynamic, it can adapt to diverse inputs or elasticity (scale in/out at runtime).

Work stealing. In parallel computing, work stealing is a commonly used strategy [29] for the load balance problem. In a work stealing scenario, we usually put the work items (*a.k.a.* tasks) on a queue, and each worker has a queue of tasks to perform. Each task consists of a series of work that has to be executed sequentially. And workers are expected to consume the tasks from their associated queues. When a worker runs out of tasks, it looks at the queues of the other workers and "steals" their tasks for their queues.

III. The Dynamic Load Balance Problem

To solve the DLB problem, we have to balance the workload of each worker at runtime. As described in Section II, the frontiers can be regarded as basic work items to process in each iteration. Intuitively, by stealing parts of frontiers, the GPU with a lower workload can spend free cycles to help others with heavier workloads to avoid starvation, thus reducing the end-to-end processing time. We call such stealing strategy frontier stealing (short for `FSteal`). The challenge of `FSteal` is to decide which GPUs to be the victim and how many items to steal. Note that the neighbors of the stolen frontier are still on the original device, which means the graph topology (adjacency list) will be accessed remotely via NVLink in computation.

Example 2: As shown in Figure 3, *w.l.o.g.*, consider the stealing policy for BFS algorithm under a 2-way partition. We partition the graph into two fragments, vertices $0 \sim 3$ and associated edges are assigned to F_0 while others are assigned to F_1. In the second iteration, the BFS root vertex 0 (short for v_0) activates v_1, v_2, v_3, and v_5. GPU$_0$ processes the v_1, v_2, v_3 with 8 edges, while GPU$_1$ processes v_5 with only 2 edges. With frontier stealing, we can migrate v_3 from GPU$_0$ to GPU$_1$ lest GPU$_0$ becomes a straggler. Although GPU$_0$ and GPU$_1$ both have 5 edges and 2 vertices to process in this example, we have to access the neighbors of v_3 over NVLink instead of the local memory bus, which will add cost to computation kernels. Thus the plan of `FSteal` shown in Figure 3 is not optimal.

A. Formulation of Frontier Stealing

The policy of `FSteal`. A frontier stealing policy $\mathcal{P}^k_{\texttt{FSteal}}$ under a n-way partition is an $n \times n$ matrix where $s^k_{ij} \in \mathcal{P}^k_{\texttt{FSteal}}$ means worker j should steal s^k_{ij} vertices from worker i, in the k-th iteration. For the stolen frontier s^k_{ij}, we have $s^k_{ij} \subseteq f^k_i$. Therefore, GPU$_j$ has to access remote data (*e.g.,* the weights on edges) on GPU$_i$ when it performs the `Gather` as discussed in Section II. The goal of `FSteal` is to generate an optimal $\mathcal{P}_{\texttt{FSteal}}$ to balance the runtime of each GPU with low overhead. To avoid splitting the adjacency list which will introduce additional atomic operations, we select a group of vertices associated with required number of edges.

Quality of $\mathcal{P}_{\texttt{FSteal}}$: The target of `FSteal` is to balance and minimize the runtime of the current iteration. The runtime of GPU$_i$ completing its work in the k-th iteration is denoted by T^k_i. Then `FSteal` aims to minimize $\max^n_{i=1} T^k_i$, *i.e.,* to minimize the completion time of stragglers. Since the numbers of associated edges of different vertices vary significantly, we try to balance the edges that need to process in the k-th iteration. However, simply keeping GPUs to process the same number of edges may not be optimal, as shown in Example 2.

Optimization objective. We consider balancing the edges instead of vertices because the variance of vertices is much larger than edges, since the edge distribution of vertices is usually skewed in real-world graphs. To generate $\mathcal{P}_{\texttt{FSteal}}$, we should decide an $n \times n$ touched edges matrix X where $x_{ij} \in X$ is the number of associated edges of s_{ij}. Note that $\mathcal{P}_{\texttt{FSteal}}$ for GPU$_i$ and GPU$_j$ only steals the status of frontiers, and the stolen frontiers s^k_{ij} are only valid in the k-th iteration. In other words, the effect of `FSteal` is temporary, thus we can consider the `FSteal` only within the current iteration without worrying about the side effect for later iterations.

Given a graph G under an n-way graph partition $P(n) = (F_i, \cdots, F_n)$, and a graph algorithm $\mathcal{A}$. In the k-th iteration, we define $\mathcal{L}_k = \{l^k_1, \cdots, l^k_n\}$ to be the set of current active edges in each GPU, where each $l^k_i \in \mathcal{L}_k$ is the number of edges resident on worker i. In other words, the vertices of f^k_i are associated with the l^k_i edges. Since edges in different workers may vary, we define an $n \times n$ average cost coefficient matrix C, where the $c_{ij} \in C$ means the cost for worker j to process one edge resident on worker i, we defer the discussion of the cost coefficient matrix to Section III-B. To balance the execution time of workers, the optimization target of the work stealing problem is to:

$$\min_{ArgsX} \max_{j=1}^{n} \sum_{i=1}^{n} c_{ij} x^k_{ij}$$
$$s.t. \begin{cases} \sum\limits_{j=1}^{n} x^k_{ij} = l^k_i \\ x^k_{ij} \in \{0, 1, \cdots, |E_i|\} \end{cases} \tag{1}$$

Theorem 1: The DLB problem is NP-hard.

Proof: The proof of NP-hardness of DLB problem can be reduced to the linear programming problem. First, we let $z \geq \sum_{i=1}^{n} c_{ij} x_{ij}^{k}$, thus the DLB problem is a mixed integer linear programming (MILP) problem, which is NP-hard [30].

Discussion: We formulate a single superstep of graph algorithm instead of the accumulated cost of multiple supersteps mainly for two reasons: (1) The stealing of the entire process is hard to formulate because we can not estimate the total flow of each lane in the NVLink network due to the irregularity of graph topology. (2) The `FSteal` only processes the stolen edges while the frontiers are still generated on original GPUs, thus the min-max problem of each iteration is independent.

B. Cost Model of Frontier Stealing

The goal of the cost model is to estimate the cost coefficient matrix C described above. When stealing s_{ij}^{k} vertices to GPU$_j$ from GPU$_i$ in the k-th iteration, GPU$_j$ should process x_{ij}^{k} edges in `Gather`. The c_{ij} here means the cost of GPU$_j$ to process one edge resident on GPU$_i$, which includes the communication cost and the computation cost. More specifically, given a stealing policy $\mathcal{P}_{\texttt{FSteal}}$, we can estimate the cost of $\mathcal{P}_{\texttt{FSteal}}$ under an n-way partition of a graph G in terms of a communication cost function $h_{\mathcal{P}}$ and a computation cost function $g_{\mathcal{P}}$.

Cost model. Let $\mathcal{W}_i = \{w_1, \cdots, w_m\}$ be a set of characteristics where each $w_j \in \mathcal{W}_i$ is associated with the fragment F_i in GPU$_i$ (*e.g.,* Table I). Then, two functions h and g are defined over $\mathcal{W}$, which estimates the communication cost $h(\mathcal{W}_i)$ and computation cost $g(\mathcal{W}_i)$, respectively. The estimated communication time and computation time of GPU$_j$ under the policy $\mathcal{P}_{\texttt{FSteal}}$ is denoted by $E^{h}_{\mathcal{P}_{\texttt{FSteal}}}(j)$ and $E^{g}_{\mathcal{P}_{\texttt{FSteal}}}(j)$, respectively. Then the cost of $\mathcal{P}_{\texttt{FSteal}}(j)$ can be estimated as:

$$E_{\mathcal{P}_{\texttt{FSteal}}}(j) = E^{h}_{\mathcal{P}_{\texttt{FSteal}}}(j) + E^{g}_{\mathcal{P}_{\texttt{FSteal}}}(j). \tag{2}$$

We then show how to get $E^{h}_{\mathcal{P}_{\texttt{FSteal}}}(j)$ and $E^{g}_{\mathcal{P}_{\texttt{FSteal}}}(j)$:

(1) The communication time $E^{h}_{\mathcal{P}_{\texttt{FSteal}}}(j) = \sum_{i=1}^{i=n} h(\mathcal{W}_i) * x_{ij} = \sum_{i=1}^{i=n} \frac{x_{ij}}{B_{ij}}$ where B_{ij} is the bandwidth between GPU$_i$ and GPU$_j$ (when $i = j$, B_{ij} is the local memory bandwidth), which can be evaluated via micro benchmark. Intuitively, processing local edges is faster than processing remote edges, and processing remote edges between GPUs connected by two NVLinks is faster than GPUs connected by only one NVLink.

(2) The computation time $E^{g}_{\mathcal{P}_{\texttt{FSteal}}}(j) = \sum_{i=1}^{i=n} g(\mathcal{W}_i) * x_{ij}$: This part estimates the time of computation if we fetch all remote data into local at the cost of $E^{h}_{\mathcal{P}_{\texttt{FSteal}}}(j)$. The runtime of the computation lies mainly in the memory access of the GPU, and the atomic operations brought by the update operations.

Learning cost coefficient. Now we can see $c_{ij} = \frac{1}{B_{ij}} + g(\mathcal{W}_i)$. Next we show how to estimate $g(\mathcal{W}_i)$ in a learning approach.

Metric variables. We then select some features of the vertices as metric variables $\mathcal{W}$, which affect the behavior of the computation (*e.g.,* atomic, memory access pattern), as shown in Table I. We use the following metric variable set: $\mathcal{W} = \{\overline{d^{+}_{F_i}}, \overline{d^{-}_{F_i}}, r^{+}_{F_i}, r^{-}_{F_i}, G(F_i), H_{er}(F_i)\}$. For instance, the $d^{+}_{F_i}$ (resp. $d^{-}_{F_i}$) determines the number of incoming (resp. outgoing) neighbors that v may access during computation; $r^{+}_{F_i}$ (resp. $r^{-}_{F_i}$) described the diversity of edges; while $G(F_i)$ (resp. $H_{er}(F_i)$) described the distribution of edges.

TABLE I: The characteristics of current frontier f_k.

Symbol	Define	Description
$\overline{d^{+}_{F_i}}$	$\lvert\{u\lvert < u, v >\in E_i\}\rvert$	The average in-degree of F_i
$\overline{d^{-}_{F_i}}$	$\lvert\{u\lvert < v, u >\in E_i\}\rvert$	The average out-degree of F_i
$r^{+}_{F_i}$	$\max\limits_{u\in V_i} d^{+}(u) - \min\limits_{u\in V_i} d^{+}(u)$	The range of in-degree of F_i
$r^{-}_{F_i}$	$\max\limits_{u\in V_i} d^{-}(u) - \min\limits_{u\in V_i} d^{-}(u)$	The range of out-degree of F_i
$G(F_i)$	$\frac{2\sum_{u=1}^{\lvert V\rvert} ud(u)}{\lvert V\rvert \sum_{u-1} \lvert V\rvert d(u)} - \frac{\lvert V\rvert+1}{\lvert V\rvert}$	Gini coeffiecnt [31]
$H_{er}(F_i)$	$\frac{1}{\ln\lvert V\rvert} \sum\limits_{uinV} -\frac{d(u)}{2\lvert E\rvert} \ln \frac{d(u)}{2\lvert E\rvert}$	Degree distribution entropy [31]

Model learning. Given the metric set $\mathcal{W}_i$ described above, we model $g(\mathcal{W}_i)$ as a polynomial function. The training samples are denoted as $[W_i, t_i]$, which is extracted from the running log of $\mathcal{A}$, with the observed computational cost t_i acts as ground truth. We first run a given graph algorithm $\mathcal{A}$ on real-life and synthetic graphs to collect training set $\mathcal{D}_g$. Then we train the model with the stochastic gradient descent (SGD) algorithm [32]. SGD has been widely adopted for similar tasks in previous work, such as estimating running time of a superstep in ADP [11] and selecting the best optimization variant in GSwitch [33]. SGD has a solid theoretical basis and shown high performance and accuracy on diverse applications [34], [35]. As shown later in Section VI Exp-7, the SGD with the learning model already achieves reasonably good accuracy and performance. Note that the learning of the cost coefficients is a standalone part of `FSteal`. Other training methods that balances the efficiency and accuracy (*e.g.,* Adam) can be dropped in to replace the SGD as well. Using root mean squared relative error (RMSRE) [36] as loss function, the learning objective for $g(\mathcal{W}_i)$ is written as:

$$\min_{\Gamma} \sqrt{\frac{1}{\mathcal{D}_g} \sum_{[W_i, t_i] \in \mathcal{D}_g} \frac{(g(W_i) - t_i)^2}{{t_i}^2}} \tag{3}$$

Where Γ is weight parameters of the polynomial function. The reason for implementing $g(\mathcal{W}_i)$ as a polynomial function is twofold. (1) Polynomial regression has been proven to be effective in predicting computational cost in theory [37], which means that it can approximate any continuous function defined on a closed interval. (2) The polynomial model is more explainable compared with other black-box ML models [38], it gives the cost expression and shows which variable in $\mathcal{W}$ contributes most to the cost.

The cost coefficients have some degree of portability because graph algorithms usually have commonalities and they can be optimized under an unified abstraction [39], [14], [13], [40]. In this paper, we adopt the GAS abstraction and frontier is our core data structure. In each iteration of $\mathcal{A}$, we scan the associated edges of the current frontier and apply updates to them, which can be estimated based on the characteristics of the current frontier (shown in Table I). For example, given the same frontier for both BFS and SSSP algorithms, although they may activate totally different neighbors for the next iteration, the number of touch edges and the race condition are similar. Anyway, in the optimal case, we can train the model separately for each algorithm to get the optimal decision effect at the cost of longer train time and low flexibility. In this paper, we run all the BFS, PR, SSSP, and CC algorithms on 624 graphs from [41], and treat the running log of each iteration as independent training samples.

C. Frontier Stealing Algorithm

Algorithm 1: The frontier stealing algorithm

```
Input : Cost coeffiecnt Matrix $C$, Workload $\mathcal{L}$
1  Initialization $X, D, F \leftarrow 0$ ;
2  Initialization $z \leftarrow 0$ ;
3  build MILP solver target to $min(z)$;
4  with restriction:
5      R1: $z \geq \sum_{i=1}^{n} c_{ij}x_{ij}^{k}$
6      R2: $\sum_{j=1}^{n} x_{ij}^{k} = l_i^k$
7      R3: $x_{ij} \in \{0, 1, \cdots, |E_i|\}$
8  $X$ = MILPSolver($C$, $\mathcal{L}$, $m$);
   // select $s_{ij}$ according $x_{ij}$
9  for $i \in \{1, \ldots, m\}$ do
10     for $v \in Vi$ do
11         D[v] = $d^{+}(v)$ ;
12     end
13     D = PrefixSum(D) ;
       // $X_i$ is $i$-th row of $X$
14     F = PrefixSum($X_i$) ;
15     F = SortedSearch(F, D) ;
16     for $j \in \{1, \ldots, m\}$ do
17         Send $\{v_i | F[j-1] \leq i \leq F[j]\}$ to GPU$_j$ ;
18     end
19 end
```

As discussed above, for the distributed frontier f_i^k in the k-th iteration on GPU$_i$, the `FSteal` algorithm tries to generate the matrix X^k, while the number of touched edges x_{ij}^k means the edges resident on GPU$_i$ that GPU$_j$ should process in `Gather`. Specially, when $i = j$, x_{ij} means the amount of edges that GPU$_i$ has to process locally. Intuitively, we should minimize the execution time of the slowest worker in each iteration. To solve the min-max problem, we add a restricted condition $z \geq \sum_{i=1}^{n} c_{ij}x_{ij}^k$, and the optimization target of DLB problem is to minimize z, which can further be formalized into a mixed integer linear programming (MILP) model. The MILP solver is able to find the touched edges matrix which implies all the GPUs in the system complete their work at the same time. Since the number of touched edges x_{ij}^k in the exact solution of MILP problem may not be an integer, thus we round up the results. The `FSteal` algorithm is shown in Algorithm 1.

As discussed in Section III-B, we first train a model offline to estimate the cost coefficient matrix C among all the GPU pairs. GPU pairs may communicate with each other via different numbers of lanes of NVLink, thus some GPU pairs incur higher overhead than others. For each iteration, we calculate the number of edges to be processed in each GPU as their workload $\mathcal{L}$. In line 3-7, we set the optimization target for the MILP solver. In GUM, we use SCIP [42] solver to solve the MILP problem. After calculating the matrix X^k, we have to select which vertices to be sent to meet the requirement of X^k. m here is the number of GPUs involved in the computation. We calculate the prefix-sum of the out-degree of frontiers and run a sorted search to find consecutive vertices as the stealing policy $\mathcal{P}_{\texttt{FSteal}}$ to be sent to other GPUs, as shown in line 9-18.

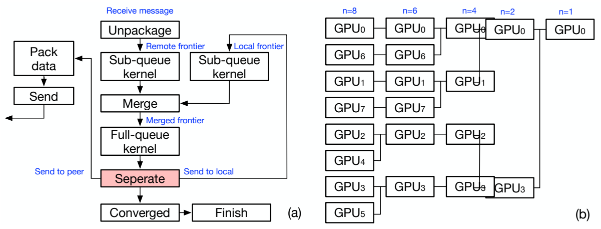

Fig. 4: (a) Gunrock kernels. (b) Reduction tree.

Time complexity analysis. For each iteration, we have to run the `FSteal` algorithm to generate $\mathcal{P}_{\texttt{FSteal}}$. Since the MILP problem is NP-Hard, so it can't be solved in polynomial time unless $P = NP$. However, MILP can certainly be solved in exponential time by branch and bound. For each iteration, we have to run the `FSteal` algorithm to generate $\mathcal{P}_{\texttt{FSteal}}$. We assume the MILP solver has a time complexity of $e(m)$, which is beyond polynomial time. After obtaining the MILP result X, the prefix-sum of the out-degree of frontiers can be calculated in $O(|V_i|)$ and the time complexity of the sorted search is $O(m\log(|V_i|))$ based on binary search. To sum them up, the time complexity of each iteration is $e(m)+\sum_{1\leq i\leq m}(O(|V_i|)+O(m\log(|V_i|))) = O(e(m)+|V|+m\log(|V|))$. Note that the complexity of the solver is only related to the number of GPUs, thus it is acceptable because the number of GPUs is small (*e.g.,* 8) for a modern GPU server. We measured the overhead of our `FSteal` algorithm, and the result is shown in Section VI-C.

IV. The Long Tail Problem

The LT problem hurts the performance mainly due to the synchronization overhead. Figure 4 (a) shows the kernels that Gunrock will invoke in one iteration. The overhead of synchronization consists of latency of kernel launching, exchange of message size with peers, and manipulating of the communication buffers. For example, the `separate` step in Gunrock will split the newly activated vertices into several bins, each for one destination GPU. Actually, the more GPUs involved, the higher the cost of synchronization. A straightforward solution to reduce the overhead is to evict some GPUs out of the processing if that does not introduce too much additional work to other GPUs. To free a GPU$_i$ from the following computation, another GPU must take ownership of the fragment F_i that resides on GPU$_i$, *a.k.a.* the ownerships of fragments on the evicted GPUs will be taken by the remainders. We call this kind of stealing the ownership stealing (short for `OSteal`), which can solve the LT problem described in Example 1.

Example 3: Figure 3 shows a 2-way partition for a BFS algorithm, GPU$_0$ has 8 edges to process while GPU$_1$ has only 2 edges to process. In `FSteal` our goal is to balance the workload by stealing some edges from GPU$_0$ to GPU$_1$. We have discussed a possible FSteal policy in Example 2 that GPU_0 processes v_0, v_1 while GPU_1 processes v_3, v_5. However, In OSteal, our goal is to reduce the synchronization overhead which is related to the number of involved GPUs. To this end, we make GPU_0 take the ownership of GPU_1, it means that GPU_0 will handle all the computation in Fragment_1 in the following iterations and GPU_1 will not participate in synchronization anymore. As a result, GPU_0 will processes v_1, v_2, v_3, and v_5. The difference between FSteal and OSteal in this Example is that FSteal considers the synchronization overhead as negligible while OSteal does not. Thus FSteal focuses on balancing the computation time whereas OSteal focuses on reducing the synchronization overhead.

A. Formulation of Ownership Stealing

The policy of OSteal. An ownership stealing policy $\mathcal{P}_{\texttt{OSteal}}$ is an n-dimension vector, *i.e.,* $o_j \in \mathcal{P}_{\texttt{OSteal}}$, while $o_j = i (1 \leq i, j \leq n)$ means that GPU_i steals the ownership of the whole fragment F_j located in GPU_j. The primary goal of OSteal is to reduce the synchronization overhead by excluding some GPUs from the communication group, at the cost of increasing workload in other GPUs. Unlike FSteal, OSteal is a long-term behavior, when OSteal occurs between GPU_i and GPU_j, not only the workload of the current iteration will change, but also all the subsequent messages that should be sent to the GPU_j in turn will be forwarded to the new owner of F_j. Thus, when GPU_i steals the ownership of F_j in GPU_j, GPU_j will have no workload to do in following iterations unless it is enabled again.

Reduction Tree. To search for the optimal $\mathcal{P}_{\texttt{OSteal}}$, we have to enumerate all the possible causes of $\mathcal{P}_{\texttt{OSteal}}$, which has $\sum_{i=1}^{n-1} C_n^i i^{n-i}$ options (it is not $\sum_{i=1}^{n-1} C_n^i$ because who steals the ownership also matters) in an n-way partition, which is a huge number even n is small. Fortunately, we can reduce the overhead with the properties of OSteal. Looking at the NVLink topology shown in Figure 2, we notice that the communication links in GPUs have a certain degree of equivalence, *i.e.,* removing (GPU_2, GPU_3) or (GPU_4, GPU_5) from the network will lose an equal amount of bandwidth. Furthermore, the runtime is dominated by synchronization overhead when OSteal occurs, since the goal of OSteal is to reduce the synchronization overhead when the computation cost is comparable to the former. Therefore, the performance behaviors of equivalent policies are similar. Based on the above observations, we can steal the ownership in a fixed order to simplify the decision process of OSteal. As shown in Figure 4 (b), the OSteal of n GPUs can be organized into a reduction tree, according to their bandwidth. In other words, we want to keep the residual network to have the largest aggregated bandwidth. For example, if a $\mathcal{P}_{\texttt{OSteal}}$ indicates that 2 GPUs are better than 8 GPUs, then leaving (GPU_0, GPU_3) is much better than leaving (GPU_0, GPU_7).

Quality of $\mathcal{P}_{\texttt{OSteal}}$: By trading off between parallelism and synchronization overhead with OSteal, we can achieve better performance. The quality of $\mathcal{P}_{\texttt{OSteal}}$ can be evaluated similarly to $\mathcal{P}_{\texttt{FSteal}}$, by adding the synchronization overhead $\hat{T}_i^k$ into consideration. OSteal aims to minimize $\max_{i=1}^{n}(T_i^k + \hat{T}_i^k)$. We can still use the FSteal algorithm described in Section III to estimate T_i^k for each GPU_i . With the reduction tree, the search space is small, thus we can enumerate all the cases and generate the optimal $\mathcal{P}_{\texttt{OSteal}}$ with negligible overhead.

For a given policy $\mathcal{P}_{\texttt{OSteal}}$, we consider the cost of a single iteration after stealing, which can be divided into two parts: (1) the cost of kernels; (2) the cost of synchronization overhead.

(1) The cost of the kernel comes from distributed computation and communication. It is decided by the last worker that completes the computation kernel. Given the $\mathcal{P}_{\texttt{OSteal}}$, the minimum value of this cost is the optimization target (denoted by z) of the frontier stealing problem in Section III, thus could be estimated by the MILP solver.

(2) The cost of synchronization overhead includes the cost of kernel launching, inevitable data movements, preparing message buffers, etc. They are essential in every iteration, and roughly in proportion to the cluster size, thus we can estimate $\hat{T}_i^k = p * m$, where m is the current number of workers, and p is a parameter that can be estimated during previous iterations.

Then, the total cost of OSteal can be estimated as:

$$E_{\mathcal{P}_{\texttt{OSteal}}} = z + p * m. \tag{4}$$

When the workload in each GPU is extremely low, especially in the late stage of the algorithm, the synchronization overhead dominates the runtime, which is the second part of Equation 4. In this situation, we can fold the GPU scale to reduce the total overhead. For example, reduce the number of workers from $m = 8$ to $m = 4$. On the other hand, when the workload is high, the first part accounts for the vast majority of Equation 4, thus a larger GPU scale will lead to higher efficiency. Therefore, more workers should be enabled under these circumstances. In other words, ownership stealing makes a trade-off between parallelism and efficiency. And Equation 4 decides the most suitable GPU scale.

B. Ownership Stealing Algorithm

The ownership stealing algorithm is shown in Algorithm 2. We first enumerate the resulting cluster size m which satisfies $1 \leq m \leq n$. And then, the ownership vector $O(m)$ is generated based on the reduction tree discussed above. After that, the MILP solver of Algorithm 1 is utilized to obtain the minimum value of the computation kernel cost z. We will discuss how to revise the cost coefficient matrix of FSteal more specifically in Section V. Together with other costs estimated as $p * m$, the total cost of an iteration under the current OSteal policy is estimated. Among all possible policies, the policy with the lowest cost will be selected.

In the initial stage of an algorithm, all the n workers are involved, with each worker holding one fragment under the n-way partition of graph G. In each of the following iterations, this ownership stealing algorithm runs on the coordinator. Once the ownership stealing is actually decided to be executed (*i.e.,*, the most suitable m decreases or increases), the resulting policy will be scattered to each worker. And after the ownership stealing is completed, a new set of workers R is formed to participate in later processing. For example, when executing the BFS algorithm on a sparse graph, the workload may be low in the early stage, so we shrink the GPU scale to make only a few workers participate in the processing. As the workload increases, more workers will be utilized, with the stolen ownership given back. And in the late stage of the algorithm, the workload is extremely low, thus the GPU scale shrinks again to reduce the total overhead.

Time complexity analysis. Modern MILP solvers are highly efficient. For each cluster size m, the stealing policy and worker set can be directly generated from the reduction tree in time $O(1)$. After that, the algorithm takes $e(m)$ for calling the MILP solver (line 6), $O(mn)$ for obtaining the kernel value z (line 7), and $O(1)$ for recording the lowest cost. Hence, the total time complexity of obtaining the minimum kernel cost is $O(e(n))$ (noting that $\sum_{m\leq n} O(e(m)) = O(e(n))$ and $O(e(n)+n^3+n) = O(e(n))$). Similarly to frontier stealing algorithm, the owner stealing algorithm is acceptable as its time complexity is only related to the number of GPUs. What's more, when the cluster size decreases, the time for executing this `OSteal` algorithm once will further decreases.

Algorithm 2: The ownership stealing algorithm

```
   Input : Cost coeffiecnt Matrix $C$, Workload $\mathcal{L}$, Reduction
           Tree $T$, $p$
1  Initialization $O \leftarrow 0$ ;
2  Initialization $Cost \leftarrow INF$ ;
   // enumerate the number of workers
3  for $m \in \{1,\ldots,n\}$ do
4      Generate the stealing policy $O(m)$ based on $T$;
5      Generate the worker set $R(m)$ based on $T$;
       // call a revised MILP solver of Algorithm 1
          to get $min(z)$
6      $X$ = MILPSolver($C$, $\mathcal{L}$, $R(m)$, $m$);
7      $z = \max_{j\in R(m)} \sum_{i=1}^{n} c_{ij}x_{ij}^{k}$ ;
8      if $z + pm < Cost$ then
9          $Cost = z + pm$ ;
10         $O = O(m)$ ;
11     end
12 end
13 for $i \in \{1,\ldots,n\}$ do
14     Set ownership of $F_i$ to GPU$_{O_i}$ ;
15 end
```

V. GUM Implementation

In this section, we describe the GUM implementation which is based on the `FSteal` and `OSteal` algorithms. We use an example to show how to solve the DLB and LT problems in the GUM workflow. At last, we discuss several optimization techniques in GUM to improve the stealing mechanism and reduce the overhead.

A. *The workflow of* GUM

The workflow of GUM is shown in Figure 5. At first, GUM loads the graph data from the disk and then partitions the graph into fragments. As Figure 5 shows, *w.l.o.g.*, the number of fragments is the same as the number of GPUs, and each

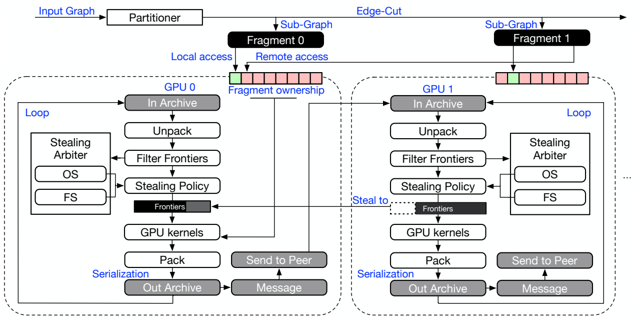

Fig. 5: The workflow of GUM.

fragment is assigned to one GPU. In a fragment, the vertices that are partitioned within the fragment are called "inner" vertices, and the destination of cross-fragment edges are also kept as "outer" vertices for message-aggregation optimization. The inner vertices of each fragment are non-overlapping, and we assume the aggregated memory of GPUs is sufficient to store the whole graph.

We assign a GUM worker with each GPU, which is responsible for conducting communication for distributed graph processing, *e.g.,* sending/receiving vertex data to/from remote GPUs and forwarding messages as a transit station for other GPUs. The worker with the lowest GPU ID works as the *coordinator*. Before each iteration, all the workers in the system perform synchronization to get the number of work items in the current iteration. Then the *coordinator* determines whether the algorithm is converged based on the workload on each GPU. The work stealing mechanisms (*a.k.a.* `FSteal` and `OSteal`) lie at the heart of GUM, which takes the communication relations among the GPUs and the communication topology as input, and output the stealing policy to reorganize the workload for each GPU. The objective of the stealing arbitrator in Figure 5 is to minimize the end-to-end execution time in the current iteration and to minimize the stealing overhead. A stealing policy describes how the workload will be transmitted between each pair of GPUs. In each iteration, the stealing policy is generated by the *coordinator* before the computation starts, and then the *coordinator* immediately broadcasts the policy to all workers.

The interaction of `FSteal` and `OSteal`. As described in Example 3, the `FSteal` focuses on the balance of workload while the `OSteal` focuses on minimizing the synchronization overhead. *w.l.o.g.*, we discuss how the `FSteal` and `OSteal` co-work in the SSSP problem described in Example 1. In each iteration, GUM first enumerates the number of involved GPUs (denoted by m), and generate the $\mathcal{P}_{\texttt{OSteal}}$ based on the reduction tree. Then GUM solves the MILP problem under the `OSteal` policy to generate an `FSteal` policy and record the estimated runtime. After exhausting all the cases (from 1 GPU to 8 GPUs), GUM chooses the `OSteal` and `FSteal` policy which minimize the estimated runtime. In the SSSP problem shown in Example 1, the first 20 iterations process a large amount of edges, and the computation time is much larger than the synchronization overhead, so the `OSteal` keeps the number of involved GPUs as many as possible and does not make decisions to steal any ownership. When the algorithm goes to the late stage and the LT perform happens, the synchronization overhead is now comparable with the computation time and such a situation may last about 200 iterations, so the `OSteal` shrinks the size of involved GPUs.

Example 4. Consider the workflow shown in Figure 5. After the graph is partitioned into n fragments, a graph algorithm will be executed in the following steps:

Step 1: Generate frontiers. Each worker will process its message buffer at the beginning of each iteration. Then it will generate the frontiers in the current iteration in each GPU.

Step 2: Ownership stealing. In order to solve the DLB and the LT problems at the same time, we allow the execution of `FSteal` and `OSteal` simultaneously during the algorithm. Since the `OSteal` needs to take over the ownership of the whole fragment, the generated frontier queue will directly be on another GPU, which has a great impact on `FSteal`. For example, if GPU$_i$ steals the ownership from GPU$_j$, then GPU$_j$ will just work like an extended memory and not participate in the following computation. Therefore, we decide on the `OSteal` policy before the `FSteal`.

Step 3: Frontier stealing. As discussed above, `OSteal` is considered before `FSteal`, and it will evict some GPUs from current iterations. To allow `OSteal` and `FSteal` to interact with each other, `FSteal` should amend the cost coefficient matrix. Denote by R as the removed GPUs after applying the `OSteal` algorithm. We should set $c_{ij} \in C_{\mathcal{P}_{\texttt{FSteal}}}$ to ∞ for each j in R, which will force the `FSteal` algorithm not to assign any frontiers to the removed GPUs.

Step 4: Processing the frontiers. After all GPUs steal what they need based on the stealing policies $\mathcal{P}_{\texttt{OSteal}}$ and $\mathcal{P}_{\texttt{FSteal}}$, the stolen frontiers (resp. ownership) have to be processed in a separated kernel, thus introducing additional overhead which is outside our cost model.

B. Optimizations

Executing `FSteal` and `OSteal` algorithms before processing the frontiers may bring the overhead. If concerns about the overhead are raised, we can alleviate the overhead of these decision process. We next show how to reduce the overhead by examples.

Example 5: Since generating a $\mathcal{P}_{\texttt{FSteal}}$ in each iteration may bring overheads, especially when GPUs have sparse frontiers or already-balanced workload to process. We can bypass the frontier stealing step if we can recognize these cases. To avoid dispensable stealing, we set two threshold values t_1 and t_2 to activate the stealing mechanism only when: (1) $\max_{i=1}^{m} l_i^k \geq t_1$, which means we have sufficient frontiers to steal in order to cover the overhead of `FSteal` algorithm. (2) $\max_{i=1}^{m} l_i^k - \min_{i=1}^{m} l_i^k \geq t_2$, Similarly, generating a $\mathcal{P}_{\texttt{OSteal}}$ also incurs overhead, since it has to enumerate the number of GPUs and run a MILP solver for each try. Fortunately, we can bypass the `OSteal` algorithm with prior knowledge.

TABLE II: Representative graphs for benchmarking.

Abbr.	Graphs	Vertices	Edges	Diameter	Domain
LJ	soc-LiveJournal1	4.85M	85.7M	13	SN
OR	soc-orkut	3.00M	213M	7	SN
SW	soc-sinaweibo	58.7M	523M	5	SN
TW	soc-twitter-2010	21.3M	530M	15	SN
CF	com-freindster	65M	1.8B	32	SN
U2	uk-2002	18.5M	524M	25	WG
AR	arabic-2005	22.7M	1.11B	28	WG
IT	it-2004	41M	1.15B	24	WG
U5	uk-2005	39.5M	1.57B	23	WG
WB	webbase-2001	118M	1.71B	379	WG
TX	roadNet-TX	1.3M	1.9M	1054	RN
CA	roadNet-CA	1.9M	2.7M	849	RN
GM	germany-osm	11M	12M	1277	RN
USA	road-USA	23M	29M	1452	RN
EU	europe-osm	50M	54M	2037	RN

Domains: SN: Social Network, WG: Web Graph, RN: Road Network.

Since the `OSteal` is usually cost-efficient when the workload is very sparse (*e.g.,* only several frontiers are activated in each worker), thus we execute Algorithm 2 only when the previous runtime is less than t_3. In GUM, t_1, t_2, and t_3 are tuning parameters, users can set them empirically to reduce the overhead incurred by the `FSteal` and `OSteal` algorithms.

Example 6: Continue with Example 5, we adopt the adaptive scheduling by triggering the stealing mechanism only when it is worth doing. However, copying the frontiers from the remote GPU to the local GPU and accessing the neighbors via NVLinks may block the computation, especially when one steals too many work items. To further reduce the overhead of stealing, GUM can cache the status of the stolen frontiers. We observed that the vertices with high in-degree are prone to be activated many times since they receive more messages than other vertices in the graph. In GUM, we call such high in-degree vertices as hub vertices, and GUM eschews the heavy remote memory access by caching the adjacency list of hub vertices in advance. Previous works [43], [5] also enabled hub vertices caching to reduce irregular memory access. We use a bitmap to mark the cached vertices, if the stolen frontiers exist in this bitmap, GUM will access the neighbors of these stolen vertices locally instead of fetching data via NVLinks. The threshold value t_4 is a tunable parameter, if $d^{-}(v) > t_4$, we will mark the vertex v as a hub vertex.

VI. Evaluation

We evaluate the performance of GUM by answering the following seven research questions:

Q1: How efficient GUM is compared to other systems?
Q2: How well does GUM scale in multi-GPU platforms?
Q3: How effective are GUM's stealing algorithms?
Q4: How much runtime overhead does stealing introduce?
Q5: How does GUM achieve its performance?
Q6: How does GUM work on different partitioners?
Q7: How effective is the cost model in GUM?

TABLE III: The evaluation time of Gum vs. other GPU graph processing libraries.

Runtime (ms) [lower is better]																
		Social Network					Web Graph					Road Network				
Alg.	**Lib.**	**LJ**	**OR**	**SW**	**TW**	**CF**	**U2**	**AR**	**IT**	**U5**	**WB**	**TX**	**CA**	**GM**	**USA**	**EU**
BFS	Gunrock	21	40	188	101	-	21	39	146	149	221	435	348	2525	3522	9088
	Groute	45	34	162	135	619	47	70	74	181	99	196	221	12163	13576	30567
	Gum	**6**	**7**	**32**	**19**	**141**	**13**	**17**	**45**	**15**	**68**	**36**	**31**	**194**	**353**	**847**
WCC	Gunrock	50	25	274	169	914	108	423	689	713	1883	25	30	253	555	1132
	Groute	14	9	273	60	**166**	29	64	97	76	136	**1**	**1.3**	**4.5**	**8.6**	**16.9**
	Gum	**6.8**	**7.5**	**52**	**27**	206	**19**	**60**	**77**	**50**	**117**	1.5	1.5	5.7	8.9	25.9
PR	Gunrock	143	134	2482	1011	12086	438	573	**1105**	1001	2804	32	42	210	441	940
	Groute	3425	9858	5780	4840	21225	776	1754	3365	4906	2963	146	138	259	345	311
	Gum	**90**	**118**	**1166**	**631**	**11850**	**129**	**211**	1512	**512**	**661**	**8.7**	**9**	**22**	**37**	**75**
SSSP	Gunrock	98	176	671	220	2296	630	564	488	921	1145	622	486	3373	5136	12669
	Groute	197	207	1651	378	6755	214	285	310	**400**	**318**	517	450	13953	10253	38012
	Gum	**45**	**62**	**195**	**95**	**2108**	**207**	**233**	**285**	561	566	**98**	**90**	**49**	**694**	**1436**

A. Experimental Setup

We compare GUM with the state-of-the-art GPU-based graph systems using following setups:

Platform. We ran all single-node experiments on our NVIDIA GPU systems, on a Linux server with 2.10GHz E5–2620 v2 Intel Xeon CPUs and 48 GB of main memory, equipped with 8 NVIDIA V100 GPUs with 32 GB of global memory. We compiled all the GPU programs using NVIDIA's `nvcc` compiler (version 10.1.0) and the `-O3` flag. The measured results in all experiments ignore the IO time.

Dataset. We chose a variety of types of graph data, some from the social network (labeled SN), the web graph (labeled WG), and the road network (labeled RN). For the social network, it is characterized by a very skewed distribution of edges, usually with some "hot" vertices having many incoming edges. For the road network, it has a very long diameter and fewer outgoing edges per vertex. And the web graph lies somewhere in middle. As shown in Table II, we chose 15 representative graphs to analyze the performance of GUM.

Baseline. GUM was compared with several state-of-the-art programmable GPU-based graph processing frameworks/libraries. According to the latest results in [44], [7], [12], Gunrock [6] and Groute [12] are available and delivered relatively better performance than the others owing to its continuous evolution. We use four typical graph applications to benchmark GUM, Gunrock, and Groute: breadth-first search (BFS), connected components (WCC), PageRank (PR), and single-source shortest path (SSSP) algorithms. All three systems are evaluated based on a random partitioner to eliminate the influence of graph partitioning methods.

B. Main Results

Exp-1: Effectiveness. To answer Q1, we compare GUM against Gunrock and Groute using 8 GPUs to highlight the overall performance, Table III shows the result. For all dataset used, GUM significantly outperforms other GPU-based graph processing systems using 8 GPUs, especially for the graph traversal applications like BFS and SSSP. Groute runs faster in WCC on road networks because it uses an asynchronous model which has advantages on long-diameter graphs [45]. The reasons that the GUM performs better than Gunrock and Groute are (1) they do not take full advantage of NVLink, and their communication implementations are not aware of the asymmetry of the communication topology. As described in Example 1, graph traversal applications like BFS and SSSP may suffer severe DLB problem in various graph datasets. However, both Gunrock and Groute ignore the NVLink topology and the dynamic load-imbalance, while GUM enables stealing thus avoiding GPU starvation (Section III). We will discuss it in detail in Exp-2. (2) GUM achieves more speedups in road-network graphs because such graphs have long-distance which are prone to have more iterations, the iterations near the convergence have insufficient workloads to cover the synchronization overhead, thus they suffer more from the LT problem. GUM can shrink the communication group size with the help of the `OSteal` algorithm (Section IV), which can improve the performance significantly in road-network graphs.

Exp-2: Scalability. To answer Q2, we compare GUM with Gunrock and Groute on five large graphs listed in Table II. We run four graph benchmarks on a server with 8 NVIDIA V100 to investigate strong scalability.

Runtime Breakdown. Figure 6 shows the runtime breakdown of GUM with different numbers of GPUs: The computation time is the GPU kernel time, such as expanding the neighbors of frontiers and updating vertex data for active vertices. The communication time comes from data motivation between GPUs, as well as the GPU starvation time for waiting for the stragglers. The serialization time is the communication overhead, including packing irregular and scattered memory access patterns into a consecutive one to facilitate the data transfer over NVLink and PCIe. The overhead time measures the auxiliary processes such as the conversion from global vertex ID to local vertex ID, and the extra computation introduced by `FSteal` and `OSteal` algorithms. GUM achieves nearly linear speedups on BFS, SSSP, and PR algorithms, up to 6.5×, 5.3×, and 7.5× speedups (8 GPUS over 1 GPU) respectively. The scalability of GUM mainly comes from: (1) the scalability is mainly bound by the computation time, thus the load-balance which GUM solved

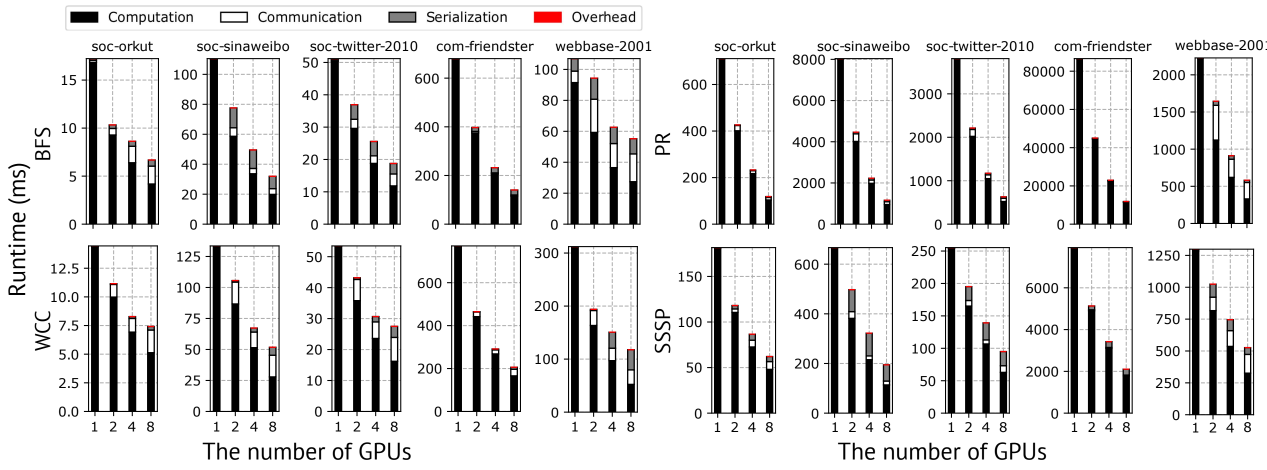

Fig. 6: The runtime breakdown and strong scalability of GUM.

with FSteal is the key to achieving high scalability. (2) GUM enables several communication optimizations such as early message aggregation and hub caching (Section V), which reduce the communication and serialization cost.

Comparison with Gunrock and Groute. Figure 7 shows the strong scalability compared with Gunrock and Groute. Both Gunrock and Groute leverage the NVLinks to accelerate the communication. However, Groute has a special design that only uses a ring from the NVLink topology as shown in Figure 2. We observed that Groute with an odd number of GPUs performs much worse than itself with an even number of GPUs because an odd number of GPUs are hard to form a ring in NVLink topology. Hence, Groute runs faster in a single GPU because of the asynchronous model, but it is hard to scale out. Gunrock and GUM are both BSP-based, however, Gunrock's implementation enabled many algorithm-specific optimizations, thus the comparison is more beneficial for them. For SSSP algorithm, Gunrock's implementation adopts an algorithm-specific "near-far" optimization [46] that runs faster on a single GPU while hard to scale out to multi-GPU. GUM does not rely on algorithm-specific optimizations, since the optimizations of specific algorithms do not fall into the ambit of GUM. Thanks to the frontier stealing and ownership stealing, GUM keeps its scalability from 1 GPU to 8 GPUs.

C. Micro Benchmark

Exp-3: Efficiency of FSteal. We show the effectiveness and overhead of FSteal in following:

Effectiveness. To answer Q3 about FSteal, we show an example of frontier stealing efficiency in Figure 8. We run the SSSP algorithm on graph *sinaweibo*. By turning on/off the frontiers stealing, we recorded the computation time of the critical iterations *i.e.,* iterations #5 and #6, which take 43 ms and 38 ms respectively. Without the FSteal, we can see the stragglers are GPU$_4$ and GPU$_5$, the faster GPU$_0$ and GPU$_1$ have to wait until GPU$_4$ and GPU$_5$ finish, wasting 72% and 67% of their cycles. By enabling the FSteal, wasted cycles can be leveraged to accelerate the stragglers. Thus it reduces the end-to-end time of iterations #5 and #6 to 32 ms and 29 ms respectively. The stall time is reduced to 4%, which means the cores are almost busy all the time. GUM can achieve good balance quality because it solves the problem following the estimate-and-reassign style that decides the workload from a holistic view in advance, while general work stealing methods often follow the peek-and-grap style which relies on the unpredictable behaviors of each worker at runtime. We also observed a counter-intuitive phenomenon that GPU$_7$ steals others while itself was stolen. This is because of the asymmetry of the NVLink topology that there may be no links between some workers, while GPU$_7$ works as a bridge to balance the faster worker and the slower worker.

Overhead. To answer Q4 about FSteal, we report the overhead of the FSteal algorithm in Table IV. In this setting, we run the SSSP algorithm on the two graphs *uk-2002* and *webbase*. Then we show the runtime of the overhead, as well as the ratio between the reduced time with FSteal and the overhead time. For example, we can save 100 ms in the SSSP algorithm on *uk-2002* with 8 GPUs. The overhead here includes copying the status of stolen frontiers, the inference of the learning model, and the runtime of FSteal algorithm. The overhead of FSteal in GUM was affordable, as it cost at most 13 ms of total runtime. This is because the feature vector can be collected efficiently without too much computation. As described in Section III, features can be collected with a scan over active vertices rather than edges. For example, the average degree can be easily obtained by averaging the degree of each frontier. As a reward, we can reduce 25.6× starvation time on average compared with the overhead time, which is worthwhile.

Exp-4: Efficiency of OSteal. We show the effectiveness and overhead of OSteal in following:

Effectiveness. To answer Q3 about OSteal, we show the switching process of the SSSP algorithm on two graphs (*webbase* and *USA*) in Figure 9. *webbase* is a typical web graph that has skewed vertices which hold excessive edges, and a long diameter which requires more iterations to propagate messages in the whole graph. The OSteal here can shrink

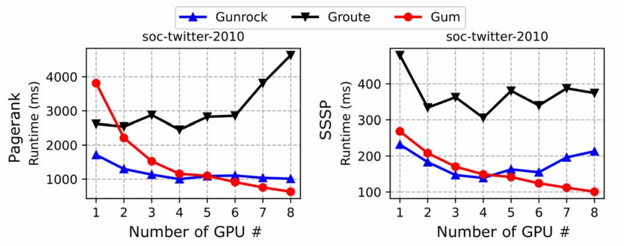

Fig. 7: The strong scalability of GUM, Gunrock, and Groute.

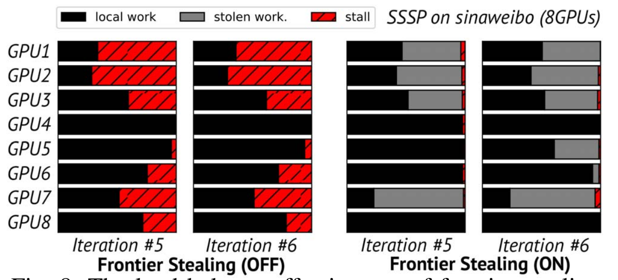

Fig. 8: The load balance effectiveness of frontier stealing.

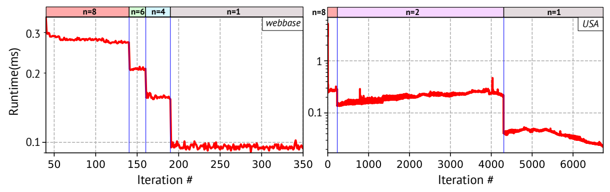

Fig. 9: The effectiveness of ownership stealing.

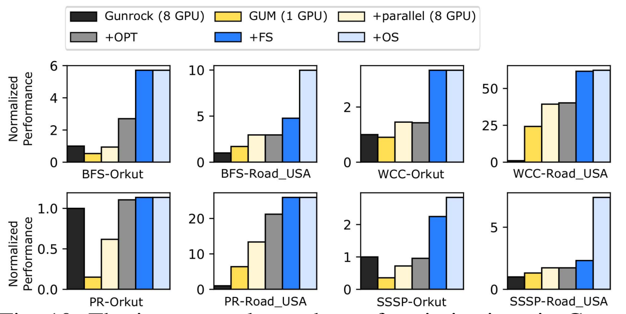

Fig. 10: The incremental speedups of optimizations in GUM.

the communication group size n to reduce the synchronization overhead as described in Section IV. At iteration #5 $\sim$ #20 of the *webbase*, the skewed vertices can activate numerous frontiers which may lead to the DLB problem that falls in the ambit of `FSteal`. At iteration #141, the `OSteal` shrinks n from `8` to `6`, and further shrinks n to `4` at iteration #162, finally shrinks it to `1` at iteration #190. Reducing the synchronization overhead in the last 300 iterations, it improves the performance by 11 %. The `OSteal` performs better in road networks like *USA*, which takes 6446 iterations to converge. By shrinking the n to `2` at #232 and to `1` at #4314, `OSteal` achieves 3.2× speedups. The reason behind these behaviors is that the synchronization time becomes a dominant factor when the LT problem happens. Although the synchronization time is below 1 ms, the accumulated overhead of 6000+ iterations in the *USA* graph may be nonnegligible. Actually, in such a situation, only several vertices are activated and some GPU even has no work to do. With the help of Algorithm 2, the `OSteal` is enabled to reduce such synchronization overhead at the cost of increasing computation workload per GPU. This explains why the runtime increases slightly when n becomes `2` in the USA dataset. Nevertheless, the runtime of $n = 2$ is still less than the runtime of $n = 8$.

Overhead. To answer Q4 about `OSteal`, we report the overhead of `OSteal` algorithm in Table IV. In this setting, we run an SSSP algorithm on two graph *uk-2002* and *webbase*. Then we show the runtime of the overhead, as well as the ratio between the reduced time with `OSteal` and the overhead time. The overhead comes from generating the `OSteal` policy based on reduction tree, and migrating the residual frontiers from victims. We observed that the overhead increases when the number of GPUs increases because the search space is larger thus we have to do more times synchronization in such a situation. Graph traversal algorithms on social graphs like *uk-2002* usually converge quickly and thus are not bothered by the LT problem, hench the benefit of `OSteal` is not that much compared with road networks. Nevertheless, the overhead of `OSteal` is usually less than 6 ms, meanwhile, we can reduce 17× synchronization time on average at the cost of this overhead.

Exp-5: Incremental speedups. To answer Q5, we illustrate the incremental speedups of a scale-free graph (*soc-orkut*) and a long-diameter graph (*road-USA*) in Figure 10. We used Gunrock as the baseline of each algorithm. We first run GUM on a single GPU without any optimization, then add optimizations one by one to study their contribution. In one GPU, the results show that the GUM baseline delivers a similar performance to that of the Gunrock implementation. The `opt` means the common intra-GPU optimizations which are both enabled in Gunrock and GUM, which is orthogonal with our work stealing techniques. Without `FSteal` and `OSteal`, we can observe that most of applications of GUM deliver similar performance compared with Gunrock and Groute. Therefore, the superiority of GUM in performance over Gunrock is mainly owing to the work stealing. Moreover, the importance of the `FSteal` algorithm on performance improvement varied from algorithm to algorithm. For traversal-based algorithms such as BFS and SSSP, the stealing method yielded an approximately 3.2× bump in performance, while for the PR, the workload does not change in each iteration, thus the effectiveness of our stealing is limited. The ownership stealing reduced the synchronization time, which is the bottleneck of road network graphs.

TABLE IV: The overhead of work stealing. [Cost (ms)]

	Frontier stealing				Ownership stealing			
	uk-2002		webbase		uk-2002		webbase	
GPUs	Cost	Ratio	Cost	Ratio	Cost	Ratio	Cost	Ratio
2	7	20×	17	31×	2.2	7×	3	26×
4	6	19×	13	38×	4	9×	4	32×
8	4	25×	11	22×	5	5×	6	23×

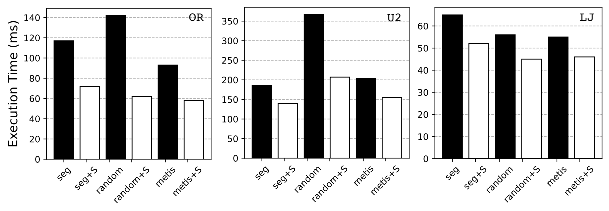

Fig. 11: The performance of GUM on different partitioner.

TABLE V: Accuracy and training time of the cost model.

Learning model	RMSRE	Training time(s)	Slowdown
Linear regression	26.7	11.1	0.54
Polynomial regression	0.33	21.7	0.93
SVR	0.21	192.3	0.94
Decision tree	0.42	25.1	0.88

Exp-6: Partitioner To answer Q6, we plot the runtime of GUM on different partitioners with or without work stealing, as shown in Figure 11. We evaluate three different types of graph partitioners: (1) `seg` is a locality-aware partitioner that tries to put adjacent vertices in one partition as well as balance the edges on each partition. (2) `random` is a trivial partitioner that assigns vertices randomly. (3) `metis` [47] is a well-known graph partitioner that minimize the edge-cut between different partitions. `+S` means that stealing is enabled. We report the performance of SSSP algorithm on OR, U2, and LJ. We observed that our stealing method can achieve 1.25∼1.63×, 1.24∼2.29×, 1.19∼1.60× on `seg`, `random` and `metis` respectively. For different graphs and different applications, the optimal graph partitioner may vary. GUM has to spend more time processing the stolen vertices under a bad partitioning scheme like `random`. However, `seg` and `metis` are not always the optimal one, `FSteal` and `OSteal` allow us to rectify the workload distribution in a suboptimal partitioning scheme.

Exp-7: Learning accuracy and effect. To answer Q7, we try different learning models to investigate their effect. We set the `degree=4` for polynomial regression, and use a `rbf` kernel for SVM regression (SVR). We use RMSRE [36] as a metric to show the difference between the exact and estimated values of $g(W_i)$. To figure out how the learning model affects the final performance, we run an SSSP algorithm two times and use the exact values of $g(W_i)$ in `FSteal` for the optimal performance, and compute the slowdown if we use the estimated version. As shown in Table V, we observed that replacing polynomial regression with a more sophisticated learning model does reduce the loss but it requires longer training time, and only provides limited performance gain (from `0.93` to `0.94`). This is because that the cost model also considers the bandwidth ($h(W_i)$) and the `FSteal` algorithm has already performed approximations to generate $\mathcal{P}_{\texttt{FSteal}}$. Considering the cost-efficiency, GUM chooses polynomial regression as our learning model.

VII. Related Work

We summarize the related work to clarify common features with other work and highlight GUM's superiority in implementing highly efficient graph analytics.

Many GPU-based graph programming frameworks have achieved high performance on a single GPU. Gunrock [6] integrated the multiple tuning strategies to implement a number of graph primitives. CuSha [23] adopted the *shards* technique and implemented the parallel-sliding-window algorithm to overcome the drawbacks associated with the CSR representation. It also provided users with a familiar GAS abstraction which was first introduced by PowerGraph[48], [14]. WS-VR [20] used warp segmentation optimization to maximize the warp efficiency. MultipleGraph [49] enable the transition between the sparse mode and dense mode to adapt to different inputs. However, these works on single-GPU optimization are limited by the restricted GPU memory. GUM breaks the barrier by implementing a high-scalability system to migrate its performance to a multi-GPU platform.

Some multi-GPU graph processing frameworks have provided their solutions to achieve high-scalability. BSP-based graph processing frameworks such as Gunrock [10] and GraphReduce [24] adopted an edge-cut graph partition method, migrating their single-GPU execution flow to multi-GPU. Although Gunrock applies a lot of optimizations, most of them are targeting the single GPU execution, such as direction-optimization and intra-GPU communication. For asynchronous multi-GPU graph processing frameworks such as Tigr [50], DiGraph [51], and Groute [12], the execution model is quite different since the asynchronous model leads to frequent communication, which are orthogonal with GUM. Furthermore, Groute only chooses one communication ring from the NVLink topology to exchange messages, which wastes the bandwidth of unselected NVLinks in modern multi-GPU servers. GUM considers the topology of communication without restricting the communication path, which is more general for different platforms.

VIII. Conclusions

In this paper, we presented GUM, an efficient multi-GPU graph analytic system with work stealing. GUM tracks the dynamic load-balance (DLB) and long tail (LT) problems which incur low GPU utilization in real world graph applications. These problems are intractable even with sophisticated graph partition technology. Thanks to the high bandwidth and low latency of NVLinks, we can rethink the work stealing based on the remote direct memory accessing. For the DLB problem, we introduce the frontier stealing algorithm that formulates the DLB problem into a MILP problem, thus it can balance the workload in each GPU with the awareness of asymmetric NVLink topology. For the LT problem, we introduce the ownership stealing algorithm that can adaptively change the communication group to trade off between parallelism and synchronization overhead. Experimental results show that GUM effectively reduces the communication time and hence improves the GPU utilization and scalability when processing large graphs. We believe our GUM design may also benefit other distributed applications or asymmetric link-topology clusters.

References

[1] S. Brin and L. Page, "Reprint of: The anatomy of a large-scale hypertextual web search engine," *Computer networks*, vol. 56, no. 18, pp. 3825–3833, 2012.

[2] J. Domke, T. Hoefler, and W. E. Nagel, "Deadlock-free oblivious routing for arbitrary topologies," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 616–627.

[3] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, "Parallel de bruijn graph construction and traversal for de novo genome assembly," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 437–448.

[4] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, "A distributed graph engine for web scale rdf data," *Proc. VLDB Endow.*, vol. 6, no. 4, p. 265–276, Feb. 2013. [Online]. Available: https://doi.org/10.14778/2535570.2488333

[5] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system." in *OSDI*, 2016, pp. 301–316.

[6] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: a high-performance graph processing library on the gpu," in *Acm Sigplan Symposium on Principles & Practice of Parallel Programming*, 2015, pp. 265–266.

[7] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel *et al.*, "Gunrock: Gpu graph analytics," *ACM Transactions on Parallel Computing (TOPC)*, vol. 4, no. 1, p. 3, 2017.

[8] webbase2001, 2021. [Online]. Available: https://law.di.unimi.it/webdata/webbase-2001/

[9] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph dsl," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276491

[10] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens, "Multi-gpu graph analytics," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 2017, pp. 479–490.

[11] W. Fan, R. Jin, M. Liu, P. Lu, X. Luo, R. Xu, Q. Yin, W. Yu, and J. Zhou, "Application driven graph partitioning," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1765–1779. [Online]. Available: https://doi.org/10.1145/3318464.3389745

[12] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-gpu programming model for irregular computations," in *ACM Sigplan Symposium on Principles and Practice of Parallel Programming*, 2017, pp. 235–248.

[13] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *ACM SIGMOD International Conference on Management of Data*, 2010, pp. 135–146.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *OSDI*, vol. 12, no. 1, 2012, p. 2.

[15] R. Nasre, M. Burtscher, and K. Pingali, "Data-driven versus topology-driven irregular computations on gpus," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 463–474.

[16] W. Fan, W. Yu, J. Xu, J. Zhou, X. Luo, Q. Yin, P. Lu, Y. Cao, and R. Xu, "Parallelizing sequential graph computations," *ACM Transactions on Database Systems (TODS)*, vol. 43, no. 4, pp. 1–39, 2018.

[17] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 599–613.

[18] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: differentiated graph computation and partitioning on skewed graphs," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, pp. 1–15.

[19] X. Shi, J. Liang, S. Di, B. He, H. Jin, L. Lu, Z. Wang, X. Luo, and J. Zhong, "Optimization of asynchronous graph processing on gpu with hybrid coloring model," in *Acm Sigplan Symposium on Principles & Practice of Parallel Programming*, 2015, pp. 271–272.

[20] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable simd-efficient graph processing on gpus," in *International Conference on Parallel Architecture and Compilation*, 2016, pp. 39–50.

[21] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus for graph applications," in *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, 2015, pp. 781–792.

[22] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *IEEE Transactions on Parallel & Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, 2014.

[23] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha:vertex-centric graph processing on gpus," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, 2014, pp. 239–252.

[24] D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan, "Graphreduce: processing large-scale graphs on accelerator-based systems," pp. 1–12, 2015.

[25] *NVLink*, January 2021. [Online]. Available: https://www.nvidia.com/en-us/data-center/nvlink/

[26] D. Yan, Y. Bu, Y. Tian, A. Deshpande *et al.*, "Big graph analytics platforms," *Foundations and Trends® in Databases*, vol. 7, no. 1-2, pp. 1–195, 2017.

[27] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, p. 103–111, aug 1990. [Online]. Available: https://doi.org/10.1145/79173.79181

[28] L. Wang, Y. Xiao, B. Shao, and H. Wang, "How to partition a billion-node graph," in *2014 IEEE 30th International Conference on Data Engineering*, 2014, pp. 568–579.

[29] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, p. 720–748, sep 1999. [Online]. Available: https://doi.org/10.1145/324133.324234

[30] D. P. Dobkin and S. P. Reiss, "The complexity of linear programming," *Theoretical Computer Science*, vol. 11, no. 1, pp. 1–18, 1980.

[31] J. Kunegis and J. Preusse, "Fairness on the web: Alternatives to the power law," in *Proceedings of the 4th Annual ACM Web Science Conference*. ACM, 2012, pp. 175–184.

[32] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[33] K. Meng, J. Li, G. Tan, and N. Sun, "A pattern based algorithmic autotuner for graph processing on gpus," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 201–213. [Online]. Available: https://doi.org/10.1145/3293883.3295716

[34] P. Zhou, J. Feng, C. Ma, C. Xiong, S. C. H. Hoi, and W. E, "Towards theoretically understanding why sgd generalizes better than adam in deep learning," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 21 285–21 296. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/f3f27a324736617f20abbf2ffd806f6d-Paper.pdf

[35] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016. [Online]. Available: http://arxiv.org/abs/1609.04747

[36] H. Park and L. Stefanski, "Relative-error prediction," *Statistics & Probability Letters*, vol. 40, no. 3, pp. 227–236, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167715298000881

[37] D. Chazan, "Introduction to approximation theory (e. w. cheney)," *SIAM Rev.*, vol. 10, no. 3, p. 393–394, jul 1968. [Online]. Available: https://doi.org/10.1137/1010083

[38] L. Huang, J. Jia, B. Yu, B.-g. Chun, P. Maniatis, and M. Naik, "Predicting execution time of computer programs using sparse polynomial regression," in *Advances in Neural Information Processing Systems*, J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, Eds., vol. 23. Curran Associates, Inc., 2010. [Online]. Available: https://proceedings.neurips.cc/paper/2010/file/995665640dc319973d3173a74a03860c-Paper.pdf

[39] J. Kepner, P. Aaltonen, D. A. Bader, A. Buluç, F. Franchetti, J. R. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, J. E. Moreira, J. D. Owens, C. Yang, M. Zalewski, and T. G. Mattson, "Mathematical foundations of the graphblas," *CoRR*, vol. abs/1606.05790, 2016. [Online]. Available: http://arxiv.org/abs/1606.05790

[40] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," *Acm Sigplan Notices*, vol. 48, no. 8, pp. 135–146, 2013.

[41] *Network Repository*, 2017. [Online]. Available: http://networkrepository.com/networks.php

[42] G. Gamrath, D. Anderson, K. Bestuzheva, W.-K. Chen, L. Eifler, M. Gasse, P. Gemander, A. Gleixner, L. Gottwald, K. Halbig, G. Hendel, C. Hojny, T. Koch, P. Le Bodic, S. J. Maher, F. Matter, M. Miltenberger, E. Mühmer, B. Müller, M. E. Pfetsch, F. Schlösser, F. Serrano, Y. Shinano, C. Tawfik, S. Vigerske, F. Wegscheider, D. Weninger, and J. Witzig, "The SCIP Optimization Suite 7.0," Optimization Online, Technical Report, March 2020. [Online]. Available: http://www.optimization-online.org/DB_HTML/2020/03/7705.html

[43] H. Liu and H. H. Huang, "Enterprise: breadth-first graph traversal on gpus," in *International Conference for High PERFORMANCE Computing, Networking, Storage and Analysis*, 2015, p. 68.

[44] *Comparison with Other Engines*, 2018. [Online]. Available: https://gunrock.github.io/docs/engines_topc.html

[45] W. Fan, P. Lu, X. Luo, J. Xu, Q. Yin, W. Yu, and R. Xu, "Adaptive asynchronous parallelization of graph algorithms," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1141–1156. [Online]. Available: https://doi.org/10.1145/3183713.3196918

[46] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel gpu methods for single-source shortest paths," in *IEEE International Parallel and Distributed Processing Symposium*, 2014, pp. 349–359.

[47] G. Karypis and V. Kumar, "Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," 1997.

[48] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *Computer Science*, 2014.

[49] C. Hong, A. Sukumaran-Rajam, J. Kim, and P. Sadayappan, "Multigraph: Efficient graph processing on gpus," in *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*. IEEE, 2017, pp. 27–40.

[50] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao, "Tigr: Transforming irregular graphs for gpu-friendly graph processing," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 622–636. [Online]. Available: https://doi.org/10.1145/3173162.3173180

[51] Y. Zhang, X. Liao, H. Jin, B. He, H. Liu, and L. Gu, "Digraph: An efficient path-based iterative directed graph processing system on multiple gpus," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 601–614. [Online]. Available: https://doi.org/10.1145/3297858.3304029