A Study of Graph Analytics for Massive Datasets on Distributed Multi-GPUs

Vishwesh Jatala
The University of Texas at Austin
Austin, Texas, USA
vishwesh.jatala@austin.utexas.edu

Roshan Dathathri
The University of Texas at Austin
Austin, Texas, USA
roshan@cs.utexas.edu

Gurbinder Gill
The University of Texas at Austin
Austin, Texas, USA
gill@cs.utexas.edu

Loc Hoang
The University of Texas at Austin
Austin, Texas, USA
loc@cs.utexas.edu

V. Krishna Nandivada
IIT Madras
Chennai, Tamil Nadu, India
nvk@iitm.ac.in

Keshav Pingali
The University of Texas at Austin
Austin, Texas, USA
pingali@cs.utexas.edu

***Abstract*—There are relatively few studies of distributed GPU graph analytics systems in the literature and they are limited in scope since they deal with small data-sets, consider only a few applications, and do not consider the interplay between partitioning policies and optimizations for computation and communication.**

In this paper, we present the first detailed analysis of graph analytics applications for massive real-world datasets on a distributed multi-GPU platform and the first analysis of strong scaling of smaller real-world datasets. We use D-IrGL, the state-of-the-art distributed GPU graph analytical framework, in our study. Our evaluation shows that (1) the Cartesian vertex-cut partitioning policy is critical to scale computation out on GPUs even at a small scale, (2) static load imbalance is a key factor in performance since memory is limited on GPUs, (3) device-host communication is a significant portion of execution time and should be optimized to gain performance, and (4) asynchronous execution is not always better than bulk-synchronous execution.

***Keywords*-Graphics Processing Units, Graph Processing, Distributed Systems, Performance.**

I. Introduction

A number of frameworks have been proposed to simplify the implementation of graph algorithms on GPUs. They can be categorized broadly into two groups: (i) frameworks for single-host multi-GPU systems, such as Gunrock [1] and Groute [2], and (ii) frameworks for multi-host multi-GPU systems, such as Lux [3] and D-IrGL [4], [5]. These systems must address a common set of challenges such as partitioning of graphs among GPUs, implementing efficient communication among GPUs, and ensuring efficient computation on each GPU.

A recent study [6] has analyzed breadth-first search on distributed GPUs using synthetic graphs. However, it is not clear how the results of the study can be extended to real-world graphs and other applications. In particular, studies on distributed CPUs [7], [8] have shown that (1) unlike synthetic power-law graphs, large real-world graphs have a relatively high diameter, so good performance on synthetic graphs does not always translate to good performance on real-world graphs, (2) different applications may perform very differently, and (3) performance is very sensitive to the policy used to partition the graph. No study of these issues has been performed on distributed GPUs. In addition, there are several computation and communication optimizations that have been shown to be useful for distributed CPUs [4], [5], but their interplay on distributed GPUs has not been studied, especially for different partitioning policies.

In this paper, we present the first detailed analysis of graph analytics applications for massive real-world datasets on a distributed multi-GPU platform. For the study, we use the state-of-the-art distributed GPU graph analytical framework, D-IrGL [4], [5]. D-IrGL is built using the Gluon communication substrate [4] and the IrGL compiler framework [9]. D-IrGL provides a variety of partitioning policies using a partitioner called CuSP [8]. It supports both bulk-synchronous and bulk-asynchronous [5] execution. We analyze a number of applications in D-IrGL using different graph partitioning policies, namely edge-balanced edge-cut (IEC/OEC) [3], hybrid vertex-cut (HVC) [10], and Cartesian vertex-cut (CVC) [11]. We analyze the performance of these applications in D-IrGL using different computation and communication optimizations. We also evaluate applications using Lux, the only other distributed graph analytical framework on GPUs, and analyze performance differences between Lux and D-IrGL.

Our study offers key lessons for designers and users of graph analytical frameworks for multiple GPUs as well as designers of graph partitioning policies.

- Cartesian vertex-cut (CVC) is critical to scale the computation to a large number of GPUs. We observe that CVC almost always outperforms the other partitioning policies on 16 or more GPUs. This is in contrast to prior studies on CPUs (with bulk-synchronous execution) [7], which found that edge-cuts were typically better at such small scale. This is important as single-host multi-GPU machines are now being designed with 16 GPUs (such as NVIDIA DGX2), but existing single-host multi-

GPU graph analytics frameworks other than D-IrGL currently do not support vertex-cuts.

- Prior studies on CPUs [7] have found that static load balancing metrics, such as the number of edges in each partition, are not strongly correlated to dynamic load balance since the number of active vertices changes unpredictably from round to round. However, our study shows that these static load metrics are important for multi-GPU execution for a different reason: they determine the amount of memory needed to store the graph in GPU memory, and imbalanced partitions may prevent the computation from running at all. Therefore statically balanced partitions are important for multi-GPU machines.

In addition, our study identifies several improvements that can be made to D-IrGL.

- Communication between devices and hosts consumes a significant portion of the execution time. Performance can be improved by overlapping communication with computation and by communicating directly between devices (using NVIDIA GPUDirect).
- In a few cases, bulk-asynchronous execution performs worse than bulk-synchronous execution. Performance can be improved by dynamically throttling the degree of asynchronous execution.

The rest of the paper is organized as follows. Section II provides background on graph analytics and multi-GPU architectures. Section III provides an overview of distributed GPU graph analytics. Section IV describes our experimental methodology. Section V presents our evaluation and analysis. Related work and conclusions are presented in Sections VI and VII, respectively.

II. Background

A. Graph Analytics

A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E. Each vertex and/or edge in the graph is associated with one or more labels representing the state of the algorithm. For the graph analytical applications studied in this paper, algorithms read and update the state in rounds. Algorithm execution ends when some termination condition is met, such as when a bound on the number of iterations is reached or when no vertex label can be updated.

State updates can be represented abstractly as an *operator* [12]. Operators are applied to active vertices in the graph, and an application reads and updates labels in a small region in the neighborhood of the active vertex. *Vertex programs* are programs in which an operator's neighborhood consists only of immediate neighbors of the active vertex. Push-style vertex programs read a vertex's label and update the labels of the neighbors of that vertex, while pull-style vertex programs read the neighbor's labels and update the label of the active vertex.

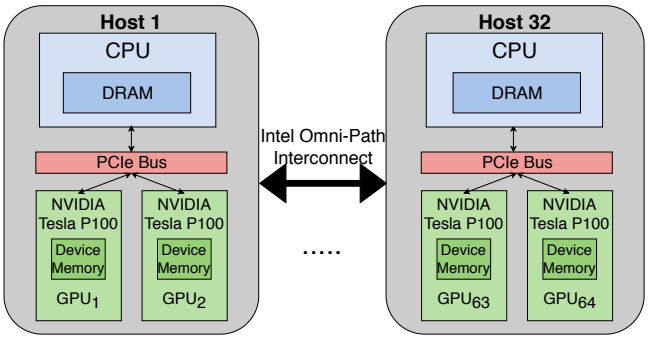

Figure 1: A typical multi-host multi-GPU setup.

B. Multi-GPU Architectures

Figure 1 illustrates the architecture of a typical multi-GPU system spanning one or more hosts connected using a network interface such as Intel Omni-Path or Mellanox FDR Infiniband. Each CPU may be connected to one or more GPUs using an interface such as a PCI Express bus. Depending on the network topology, GPUs within the same host can communicate directly (e.g., with NVIDIA GPUDirect Peer to Peer) or via the host. Similarly, GPUs from different hosts can communicate directly (e.g., NVIDIA GPUDirect RDMA technology) or via the hosts.

A typical GPU architecture consists of a set of streaming multiprocessors (SMs) that maintain several on-chip resources: registers, ALUs, shared memory, and L1 cache. The GPU also maintains a global (device) memory and L2 cache which can be accessed by any SM.

To parallelize a program on a single GPU, a programmer performs the following tasks: (1) allocate the memory on the host using *malloc* or *cudaMallocHost* (to allocate in pinned memory), (2) transfer the data from the host to the GPU device using *cudaMemcpy*, (3) perform the computation on GPU, and (4) transfer the data back from GPU to host using *cudaMemcpy*. The region of the program that needs to be parallelized on the GPU is specified using a function called *kernel*. The kernel is invoked with a specific number of thread blocks (also known as Cooperative Thread Array (CTA)) and a specific number of threads in each thread block. The number of thread blocks that reside on each SM depends on the number of available resources and the resource requirement of each thread block. The threads of a thread block are grouped into sets of 32 threads called a *warp*, and all the threads in a warp execute program instructions in a SIMT manner.

III. Graph Analytics for Distributed GPUs

In this section, we first give an overview of distributed GPU graph analytics (Section III-A) and their distributed execution model (Section III-B). We briefly describe the policies and heuristics in partitioning the graph (Section III-C). We then briefly describe communication among those GPUs

(Section III-D) and computation optimization techniques on each GPU (Section III-E).

A. Overview

To process a graph on multiple GPUs, the graph is partitioned among GPUs. *Vertices* can be partitioned among GPUs, in which case communication is required to access the labels of remote vertices. Alternatively, a partitioner can partition *edges* and create *vertex proxies* on each GPU for the endpoints of the edges mapped to it. As a result, multiple proxies could exist for the same vertex; the average number of proxies per vertex is called the *replication factor*. For each vertex in the original graph, one proxy is anointed as the *master proxy*, and the other proxies are said to be *mirror proxies*. Intuitively, master proxies are responsible for computing the canonical values of vertex labels and communicating them to mirror proxies, which act like cached copies of data from master proxies.

Each GPU processes the active vertices in its partition. Graph algorithms are resilient to stale reads, so proxy labels can be synchronized eventually rather than at every write as is done in cache-coherent architectures. Multi-GPU graph analytical frameworks transparently handle graph partitioning and communication among the GPUs and incorporate techniques to optimize computation on each GPU. Differences among frameworks arise in the ways in which they implement these functionalities. There are several single-host multi-GPU graph analytical frameworks (restricted to multiple GPUs on a single machine) like Gunrock [1] and Groute [2]. However, as far as we know, there are only two multi-host multi-GPU graph analytical frameworks: D-IrGL [4], [5] and Lux [3]. In this study, we evaluate and analyze both D-IrGL and Lux.

B. Distributed Execution Model

In multi-GPU implementations, communication is required to synchronize the state of the graph. This can be done bulk-synchronously or asynchronously.

Bulk-synchronous parallel (BSP) execution is divided into rounds. Each BSP round has a computation phase followed by a communication phase. In the computation phase, each GPU operates on its partition of the graph and updates the labels on its local proxies. At the end of the phase of local computation, a communication phase occurs in which these updates are communicated to the GPUs that need them. Most multi-GPU systems, including Lux, use BSP-style execution.

In asynchronous parallel execution, there is no notion of global rounds; instead, each GPU performs computation and communication independently. In this paper, we focus on an asynchronous model called *bulk-asynchronous parallel* (BASP) [5] execution. Execution occurs in *local* rounds, and in each local round, each GPU alternates between computation on its partition of the graph and sending or receiving messages from other GPUs. There is no global synchronization or coordination among the GPUs. D-IrGL supports both BASP and BSP execution and uses BASP by default if the benchmark can be run asynchronously.

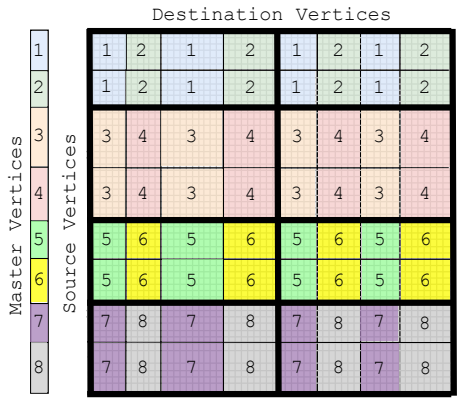

Figure 2: Cartesian Vertex-Cut (CVC) for 8 devices [7].

C. Graph Partitioning

Partitioning policies are generally divided into edge-cut and vertex-cut policies. An incoming (or outgoing) edge-cut assigns all incoming (or outgoing) edges of a vertex to the same partition. The assignment of a vertex to a partition may vary among edge-cuts. For example, an edge-balanced edge-cut [3] assigns vertices to balance the number of edges across partitions, while more complex edge-cuts such as XtraPulp's edge-cut [13] assign vertices based on neighborhood locality and load balance. We use the term IEC (or OEC) to refer to edge-balanced incoming (or outgoing) edge-cut in this paper. Unlike an edge-cut, a vertex-cut has no restriction on which partition an edge is assigned to. Many policies exist for vertex cuts, such as the hybrid vertex-cut (HVC) [10] and the Cartesian vertex-cut (CVC) [11], [4], which is a 2D cut of the adjacency matrix of the graph (see Figure 2).

Both computation and communication may be affected by the choice of partitioning policy. Computation is affected by the differences in the partitions on each host: for example, one host may have to do more work than other hosts. In BSP-style execution like in Lux, where all hosts must occasionally synchronize before continuing computation, these straggler hosts can bottleneck execution. In BASP-style execution supported by D-IrGL, the impact of stragglers on execution time may be reduced. Communication is affected by the required synchronization of proxy vertices across all hosts. The distribution of master and mirror proxies can significantly impact the communication time.

D-IrGL is the only multi-GPU graph analytical framework that supports arbitrary partitioning policies, including vertex-cuts. All other frameworks whether single-host or multi-host support only edge-cuts; in particular, Lux [3] supports only IEC. In this study, we will analyze Lux with IEC and D-IrGL with OEC, IEC, HVC, and CVC.

D. Communication Optimizations

To communicate between two devices, all existing multi-GPU frameworks transfer the message from the sending GPU to the sender host's to the receiver's host, which finally transfers it to the receiver GPU; in other words, the hosts act as a router for the device. Communication can be optimized in several ways. For example, Lux takes advantage of pinned memory to communicate updates among GPUs on the same machine. We will briefly describe two other optimizations.

1) Partition-Specific Optimizations: A reduction from mirror to master proxies followed by a broadcast from master to mirror proxies on all hosts is sufficient to synchronize proxies regardless of the partitioning policy used, but it is possible to optimize the required communication by leveraging knowledge of what the computation requires as well as the structural invariants of a partitioning policy [4]. To illustrate this, consider a graph application that only reads data from the source of an edge and only writes data to a destination of an edge. If data only needs to be read at a source of an edge, then only proxies that have outgoing edges need to have the most up-to-date value from master. Therefore, if a partitioning policy assigns all outgoing edges of a vertex to the master proxy (i.e., outgoing edge-cut), there *is no need to broadcast the master proxy's value to mirror proxies as mirror proxies do not read the value*: it suffices to reduce written values from mirror proxies to the master as only the master proxy needs to read it.

Structured vertex-cuts such as CVC can leverage their invariants to avoid all-to-all reduce and broadcast as well. To understand this, consider the adjacency matrix of the graph. Figure 2 illustrates CVC [7]. In CVC, the rows (outgoing edges) are first partitioned in a blocked fashion (labeled "masters" in Figure 2). The columns (incoming edges) are blocked the same way as rows. The matrix is then placed into a grid: in this example, it is a 4×2 grid (for 8 hosts). Finally, within each *grid row*, the blocks are distributed in a cyclic fashion to the hosts that are in that grid row. For example, the first grid row contains hosts 1 and 2, so blocks are distributed between the two in that grid row. This partitioning places mirror proxies with outgoing edges on the same grid row as its master proxy (conversely, a mirror proxy with incoming edges is on the same grid column as its master proxy). Since all proxies with outgoing edges are along the grid row, it suffices to *broadcast only to hosts in the grid row* (only proxies with outgoing edges need the updated value from the master), and since all proxies with incoming edges are along the grid column, it suffices to *reduce only with hosts in the grid column* (only proxies with incoming edges will have updates to be reduced to the master). This removes all-to-all communication [7].

D-IrGL transparently optimizes communication based on the structural invariants in the policy while also supporting arbitrary policies. As all other frameworks support only edge-cuts, they optimize communication for that specific structural invariant.

2) Update-Driven Optimization: Different proxies are updated in different rounds as the set of active vertices changes dynamically. For example, no proxies of a vertex may be updated in a given round, so no synchronization needs to occur among those proxies in that round. D-IrGL tracks updates to proxies and only synchronizes the updated values, while eliding translating between global and local addresses of the proxies during communication[1]. Tracking updates to proxies has its own overheads, so Lux synchronizes all shared data in every round (consequently, address translation is straightforward).

E. Computation Optimizations

Once the graph is partitioned and loaded on the GPU, each GPU performs computation on its portion of the graph. In each local round of BSP or BASP, each GPU visits *active* vertices in the graph, distributes the edges of those vertices among threads, and applies the operator. Graph frameworks have explored a number of implementations for computation [14], and we briefly describe some of them below.

1) Visiting Active Vertices: There are two high-level strategies for visiting active vertices in each round: topology-driven and data-driven [15]. In topology-driven execution, all the graph vertices are assumed to be active in each round, and the operator is applied to all of them. In data-driven execution, the operator is applied only to a subset of vertices where there might be work to do. Intuitively, data-driven execution is preferable if only a small subset of the vertices is active in a round.

2) Distributing Edges to Threads: Given the vertices that must be processed in a round, the computational load of applying the operator to these vertices must be distributed among the GPU threads. In power-law graphs, vertices may have very different degrees, so distributing active vertices among threads may lead to poor load balance. D-IrGL [4] supports the TWC (Thread/Warp/CTA Expansion) strategy [16] which leverages the architectural features to distribute the load. Depending on the degree of a vertex, it assigns the edges of a vertex to threads of a thread block, to threads of a warp, or to a single thread. This can handle load imbalance problems that are present within the thread block, but it does not handle the load imbalance among the thread blocks. D-IrGL by default uses an Adaptive Load Balancer (ALB) [17] strategy that assigns the edges of a very high degree vertex among thread blocks and uses the TWC to distribute the edges of all the other vertices. In contrast,

[1]Generally, a communication framework sends the global ID of the vertex along with the updated data so the receiver can convert the global ID to a local ID. Gluon elides the sending of these global addresses by memoizing the send order so that each host can identify data based on the order it is received.

Table I: Inputs and their key properties.

	rmat23	orkut	indochina04	twitter50	friendster	uk07	clueweb12	uk14	wdc14
$\|V\|$	8.3M	3.1M	7.4M	51M	66M	106M	978M	788M	1,725M
$\|E\|$	13.4M	234M	194M	1,963M	1,806M	3,739M	42,574M	47,615M	64,423M
$\|E\|/\|V\|$	16	76	26	38	28	35	43.5	60.4	37
max D_{out}	35M	33,313	6,985	779,958	5,214	15,402	7,447	16,365	32,848
max D_{in}	9,776	33,313	256,425	3.5M	5,214	975,418	75M	8.6M	46M
Approx. Diameter	3	6	2	12	21	115	501	2498	789
Size (GB)	1.1	1.8	1.6	16	28	29	325	361	493

Lux distributes the edges of each vertex (irrespective of its degree) to threads within a thread block.

IV. Experimental Setup

We first describe the multi-GPU platforms (Section IV-A) and then the multi-GPU frameworks that we study on these platforms (Section IV-B). We then describe our experimental methodology (Section IV-C).

A. Multi-GPU Platforms

We study the performance of the multi-GPU graph analytical frameworks on two experimental platforms: a multi-host multi-GPU production cluster and a single-host multi-GPU system. The machine used for the multi-host experiments is the Bridges cluster at the Pittsburgh Supercomputing Center [18]. We used up to 64 NVIDIA Tesla P100 GPUs located on 32 distributed machines with 128GB DRAM each. Each machine has 2 Intel Broadwell E5-2683 v4 CPUs with 16 cores per CPU, and they are connected through the Intel Omni-Path Architecture. Each P100 GPU has 16GB of memory. We call the single-host multi-GPU system as *Tuxedo*. Tuxedo has 2 Intel Xeon E5-2650 v4 CPUs with 12 cores and 96 GB DRAM per CPU. It contains a total of 6 GPUs: the first 4 GPUs are NVIDIA Tesla K80 and the other 2 are NVIDIA GeForce GTX 1080. Each K80 GPU has 12GB of memory, and each GTX 1080 GPU has 8GB of memory.

We use five benchmarks in our experiments: breadth-first search (bfs), weakly connected components (cc), k-core (kcore), pagerank (pr), and single-source shortest path (sssp). These benchmarks are used widely to measure the performance of graph frameworks [1], [2], [3], [4], [5], [7]. In our experiments, the vertex with the highest out-degree is used as the source vertex for bfs and sssp. For all inputs, we add randomized edge-weights. Algorithms are run to convergence. The reported execution time of applications excludes the graph loading, partitioning, and construction time[2]. Reported runtime numbers are an average of three runs.

Table I lists the inputs used in our evaluation and their properties. The rmat23 graph is a randomized scale-free graph generated using a rmat generator [19]; orkut, friendster [20], and twitter50 [21] are social network graphs; indochina04, uk07, clueweb12, uk14 [22], [21], [23], and wdc14 [24] are web-crawls.

All the graphs have a power-law degree distribution. Such graphs typically have a low diameter, but large web-crawls like uk14 have a non-trivial diameter due to long tails. The inputs in Table I are divided into three categories: small graphs are used for single-host multi-GPU experiments on up to 6 GPUs whereas medium and large graphs are used for multi-host multi-GPU experiments on up to 64 GPUs.

B. Multi-GPU Graph Analytical Frameworks

We experiment with four multi-GPU graph analytical frameworks: D-IrGL [4] and Lux [3], which support both single-host multi-GPU and multi-host multi-GPU architectures, and Gunrock [1] and Groute [2], which support only single-host multi-GPU architectures. D-IrGL and Lux are the only distributed GPU graph analytical frameworks. We chose Gunrock because it is a widely used and stable single-host multi-GPU framework and Groute because it is the only framework other than D-IrGL that supports asynchronous communication between GPUs. On Tuxedo, we evaluated all the four frameworks using up to 6 GPUs. On Bridges, we evaluated D-IrGL and Lux on up to 64 GPUs.

D-IrGL is built using the Gluon communication substrate [4] with the CUDA code generated by the IrGL compiler [9]. The IrGL compiler includes computation optimizations like the TWC [16] and ALB [17] load balancing schemes. D-IrGL provides both push-style data-driven implementations and pull-style topology-driven implementations for each benchmark. We use topology-driven execution for pr (residual based algorithm) and data-driven execution for the rest.

Lux is built using CUDA on top of the Legion [25] and GASNet [26] runtimes. We use only cc and pr in Lux as the others were incorrect or not available. They use data-driven implementation for cc and topology-driven for pr (recomputes the rank of each vertex in each round). pr on Lux does not have a run until convergence option, so we ran it for the same number of rounds executed by pr in D-IrGL. Lux uses its in-built IEC partitioning policy (we observed that it does not do dynamic repartitioning and we have notified the authors).

[2]Comparing graph partitioning time is not our focus. Graph partitioning is performed on CPUs; in practice, graphs can be partitioned once, and in-memory representations of the partitions can be written to disk. Applications can then load these partitions directly.

Gunrock [1] extends a single GPU graph processing library [27]. It includes LB load balancing scheme (that balances the edges of a vertex, irrespective of its degree, among all thread blocks) and application specific optimizations for sssp and cc. It also provides direction-optimizing traversal for bfs. It provides a set of pre-defined partitioned policies. We choose the default random partitioning strategy as recommended by Gunrock. Its pr produced incorrect output (the pagerank values did not match those produced by pr in the other frameworks), so we omit it in our evaluation. Gunrock uses data-driven execution for all the other benchmarks.

Groute [2] leverages the underlying multi-GPU network topology and low-level networking features, and it provides programming constructs for writing asynchronous multi-GPU programs. Groute uses data-driven algorithms for all the benchmarks except cc (unlike other frameworks, its cc uses a pointer-jumping algorithm) and uses the edge-cut partitioning provided by METIS [28].

C. Evaluation Methodology

We chose benchmarks implemented in the D-IrGL framework. Some of these benchmarks have been implemented in Lux, so we analyze the key performance differences between Lux and D-IrGL. Since there are many single-host, multi-GPU frameworks like Gunrock and Groute, it is also interesting to study the performance of D-IrGL on single-machine platforms. For these studies, we use smaller graphs.

As Lux supports only IEC, we first compare it with D-IrGL using IEC. Although both produce edge-balanced edge-cuts, the ones produced by D-IrGL may be different from that of Lux. Nevertheless, we modified D-IrGL to use the same partitions that are produced by Lux if it was possible (i.e., no crash as well as access to Lux partitions used). We analyzed different optimizations and variants in D-IrGL:

1) Computation optimization — TWC vs. ALB: D-IrGL supports both TWC and ALB load balancing schemes, but uses ALB by default. ALB is expected to be at least as good as TWC, and TWC is expected to be at least as good as the load balancing scheme used in Lux. The performance at scale is typically limited by communication, so the difference between them is expected to be small at scale.
2) Communication optimization — AS vs. UO: Lux synchronizes all shared (AS) proxies irrespective of whether they are updated or not. D-IrGL has an option to use AS, but by default, it tracks updates and synchronizes the updated values only (UO). UO is expected to be at least as good as AS, although tracking updates has overheads.
3) Execution model optimization — Sync vs. Async: Lux uses bulk-synchronous (Sync) execution, while D-IrGL uses bulk-asynchronous (Async) execution by default as it is expected to better than Sync.

Table II: Fastest execution time (sec) of all frameworks using the best-performing number of GPUs on the single-host multi-GPU system, Tuxedo (GPU count in parentheses).

Benchmark	Platform	rmat23	orkut	indochina04
bfs	Gunrock	0.02 (1)	0.03 (6)	0.00 (6)
	Groute	0.56 (1)	0.06 (6)	0.01 (1)
	Lux	-	-	-
	D-IrGL	(IEC) 0.03 (6)	(IEC) 0.06 (6)	(IEC) 0.01 (6)
cc	Gunrock	0.21 (2)	0.13 (6)	0.52 (6)
	Groute	0.06 (6)	0.04 (6)	0.08 (6)
	Lux	0.48 (6)	0.22 (4)	0.69 (4)
	D-IrGL	(HVC) 0.08 (6)	(IEC) 0.01 (6)	(CVC) 0.11 (6)
pr	Gunrock	-	-	-
	Groute	25.06 (2)	3.82 (6)	1.06 (4)
	Lux	1.30 (4)	2.74 (4)	4.09 (4)
	D-IrGL	(IEC) 0.5 (6)	(OEC) 2.76 (6)	(OEC) 1.33 (6)
sssp	Gunrock	0.19 (6)	0.28 (6)	0.01 (6)
	Groute	1.02 (4)	0.07 (6)	0.05 (1)
	Lux	-	-	-
	D-IrGL	(IEC) 0.03 (6)	(CVC) 0.16 (6)	(IEC) 0.02 (4)

Table III: Maximum memory usage (in GBs) across 6 GPUs of all frameworks for cc on 6 GPUs of the single-host multi-GPU system, Tuxedo (Lux uses a static memory allocation).

System	rmat23	orkut	indochina04
Gunrock	1.29	0.75	1.24
Groute	0.50	0.42	0.74
Lux	5.85	5.85	5.85
D-IrGL	0.36	0.21	0.33

Some of these optimizations have been studied separately in prior work [17], [4], [5], but it is not clear how they interact with each other and with different partitioning policies. In particular, their impact on large datasets at scale has not been studied. Studying each combination of these would be prohibitive, so we study them orthogonally. We start with the baseline variant (Var1) — TWC + AS + Sync. Then, we study ALB + AS + Sync (Var2) and ALB + UO + Sync (Var3). Finally, we study the variant with all the optimizations — ALB + UO + Async (Var4, which is the default in D-IrGL). We then examine the effect of changing the partitioning policy.

V. Experimental Results

We first present results for single-host multi-GPU frameworks (Section V-A). Then, we analyze the results for multi-host multi-GPU frameworks (Sections V-B and V-C).

A. Single-Host Multi-GPU Graph Analytics

We evaluate all benchmarks in all the frameworks using the small graphs on 1, 2, 4, and 6 GPUs of Tuxedo. Table II shows the best performance of each framework. The best-performing number of GPUs is in parentheses (less than 6 implies that the framework did not scale). Gunrock is the only framework that uses direction-optimization for bfs, and

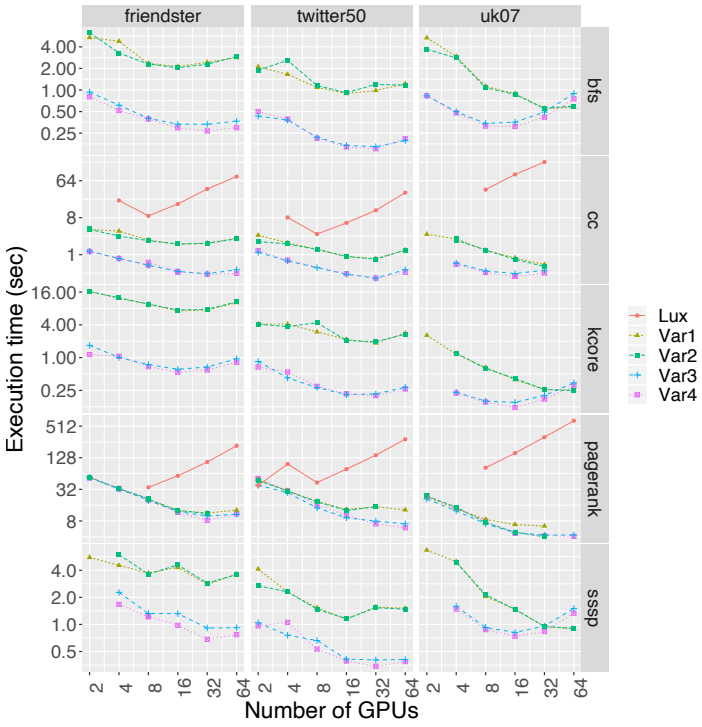

Figure 3: Strong scaling (log-log scale) of D-IrGL variants and Lux for medium graphs on Bridges (2 GPUs share a machine).

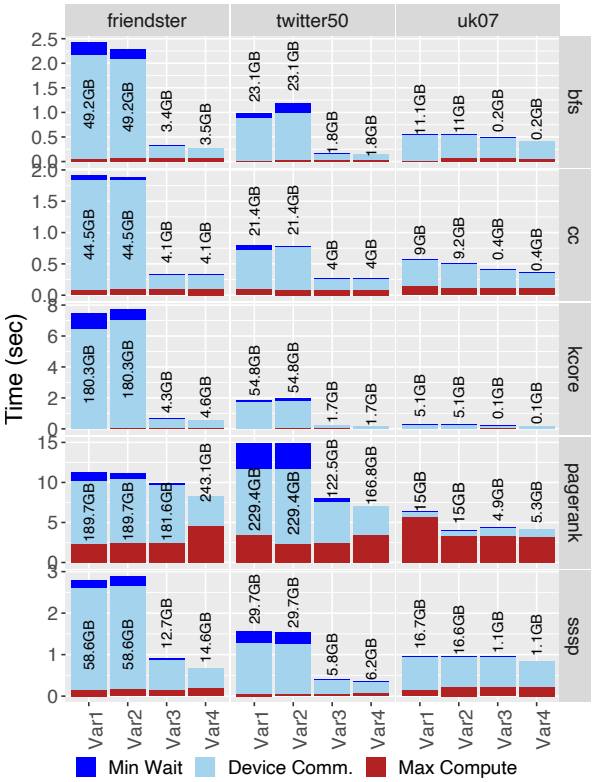

Figure 4: Breakdown of execution time of different variants of D-IrGL for medium graphs on 32 P100 GPUs of Bridges.

Groute uses pointer-jumping algorithm for cc. Consequently, they have an algorithmic advantage. Nonetheless, D-IrGL is either competitive with the other frameworks or outperforms them on this single-host multi-GPU platform. Thus, D-IrGL is a suitable framework to study the scaling out behavior of graph analytics on multi-host multi-GPUs.

We observed that we were able to run D-IrGL for the medium graphs on Tuxedo, whereas all the other frameworks ran out of memory. To analyze this, for the small graphs, we measured the memory consumed as the maximum across the 6 GPUs for each framework and we report it for cc in Table III. D-IrGL consumes less memory than the others and handles larger graphs on the same number of GPUs.

B. Multi-Host Multi-GPU Graph Analytics: Optimizations

In this subsection, we evaluate the IEC policy for both D-IrGL and Lux. We were not able to run Lux benchmarks for any of the large graphs due to memory problems[3], even on 64 GPUs. As noted earlier, the IEC partitions are the same for D-IrGL and Lux (when there were no crashes). As explained in Section IV-C, we evaluate four different variants of D-IrGL: (1) Var1 (baseline): TWC + AS + Sync, (2) Var2: ALB + AS + Sync, (3) Var3: ALB + UO + Sync, and (4) Var4 (default): ALB + UO + Async.

[3]When Lux is launched, programmers specify the estimated amount of GPU memory and zero-copy (pinned) memory needed to run it. Even with the maximum possible GPU memory and recommended zero-copy memory, it did not run.

Figure 3 shows the strong scaling behavior of D-IrGL variants and Lux programs on medium graphs using up to 64 GPUs. The missing points for both D-IrGL and Lux indicate that the benchmarks failed either due to memory limits or crashes. Figures 4 and 5 show the breakdown in execution time on 32 and 4 GPUs, respectively. We measure the computation time on each device (GPU) and report the maximum among them. We also measure the time spent on each corresponding host (CPU) waiting (blocking) to receive messages from another host and report the minimum among them. We report the rest of the execution time (maximum among devices) as the non-overlapping communication time between the device and host. In addition, we report the communication volume (in GB) on each bar. Figure 6 shows the breakdown in execution time on 64 GPUs for large graphs.

1) Lux vs. D-IrGL baseline (Var1): We analyze Lux by comparing it with the baseline Var1. This turns off specific optimizations in D-IrGL that are not present in Lux. As observed in Figure 3, Lux does not scale beyond 4 GPUs and Var1 always outperforms Lux. However, as seen in Figure 5, both Var1 and Lux perform similarly in the computation phase because both can balance load within the thread block but not among the thread blocks. Var1, even with AS, spends little time in synchronization among hosts (Min Wait Time). For 8 or more hosts, most of Lux's runtime is spent waiting.

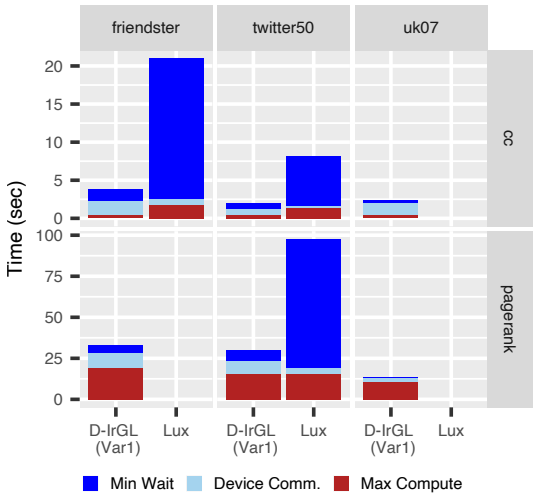

Figure 5: Breakdown of execution time of Lux and D-IrGL for medium graphs on 4 P100 GPUs of Bridges.

2) Computation Optimizations — TWC vs. ALB: D-IrGL supports ALB [17], which can dynamically balance the load among the thread blocks within the GPU. The only difference between Var1 and Var2 is that Var2 uses ALB while Var1 uses TWC. Figures 3, 4, and 6 show that Var2 performs similar to Var1 in most cases. However, for particular cases such as pagerank on uk-2007 as well as clueweb12 and uk-2014, Var2 performs better. Figure 6 shows that Var2 spends less time in computation than Var1 for pagerank on both clueweb12 and uk-2014. This is because pagerank uses pull-style algorithm that reads the incoming neighbors of a vertex to update its label, and the maximum in-degree for these inputs is huge (shown in Table I). Var1 uses TWC to distribute the edges of high in-degree vertices within the threads of a thread block, leading to a huge load imbalance among the thread blocks. Var2 uses ALB which dynamically detects the presence of thread block load imbalance and distributes the load evenly among all thread blocks within the GPU. All the other benchmarks use push-style algorithms that read the vertex's label and updates outgoing neighbors, and the same inputs have much lower maximum out-degree compared to maximum in-degree (Table I). As a result, the benchmarks do not suffer from thread block load imbalance. Consequently, the computation times (and execution times) of Var1 and Var2 are similar for the other benchmarks.

3) Communication Optimizations — AS vs. UO: D-IrGL supports tracking updates and sending only the updated values (UO). Var3 uses UO while Var2 sends all the shared values (AS). Figures 3, 4, and 6 show that Var3 outperforms Var2 for almost all cases (except uk-2007 on 64 GPUs). Var3 is faster than Var2 because not all vertices are updated in every bulk-synchronous (BSP) round, and synchronizing labels of all vertices (AS) leads to unnecessary communication. We observe that Var3 reduces host-device communication volume and time significantly in almost all cases. For uk07

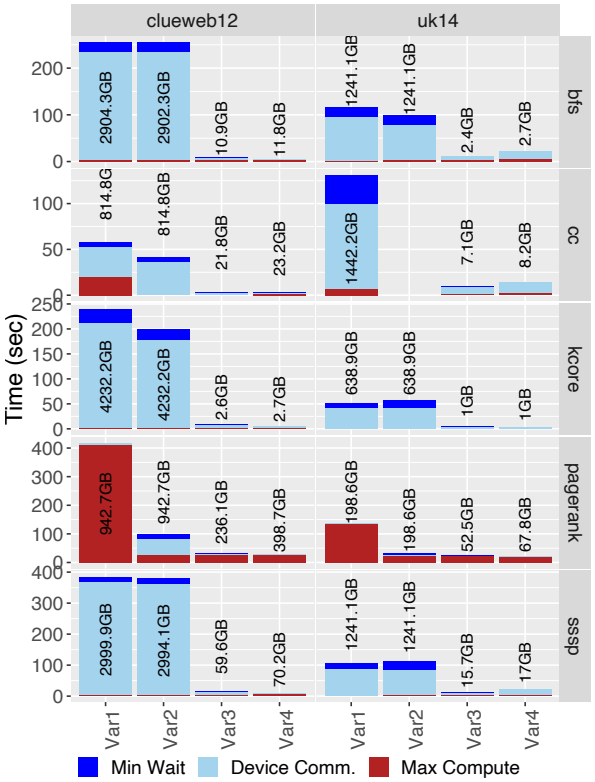

Figure 6: Breakdown of execution time of different variants of D-IrGL for large graphs on 64 P100 GPUs of Bridges.

with sssp on 64 GPUs (which executes 155 BSP rounds), we observe that the average message size was reduced from ∼2MB to ∼0.2MB when switching from Var2 to Var3. Although this is a reduction, communication of these small messages is bound by the latency, so the communication time is similar. However, UO has the overhead of extracting the updated values from the GPU (using a prefix-scan). Var3 is slower than Var2 due to this overhead for this case. To contrast this effect, consider friendster for sssp on 64 GPUs (executes 26 BSP rounds), the average message size was reduced from ∼60MB to ∼10MB when switching from Var2 to Var3. As these are larger messages, this reduction leads to a reduction in communication time that offsets the overhead of prefix-scan.

Sending only the updated values is key to reducing the communication volume and time, but there is a threshold below which the overhead of extracting the updated values outweighs the benefits of volume reduction. This threshold can be determined using microbenchmarking, and existing multi-GPU frameworks can benefit from doing this.

4) Execution Model — Sync vs. Async: D-IrGL supports bulk-asynchronous execution (Async). Var4 uses Async (the rest use Sync). Figures 3, 4, and 6 show that Var4 outperforms Var3 in most cases. Async minimizes idle time as it does not wait for messages from other hosts to perform the next round of computation. This may result in faster convergence as faster hosts can communicate the updated

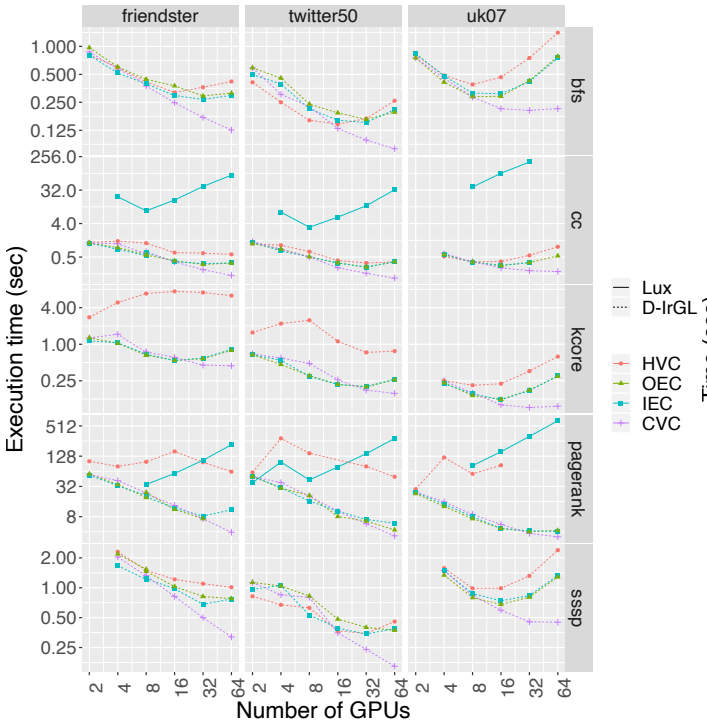

Figure 7: Strong scaling (log-log scale) of D-IrGL with different partitioning policies and Lux for medium graphs on Bridges (2 GPUs share a physical machine).

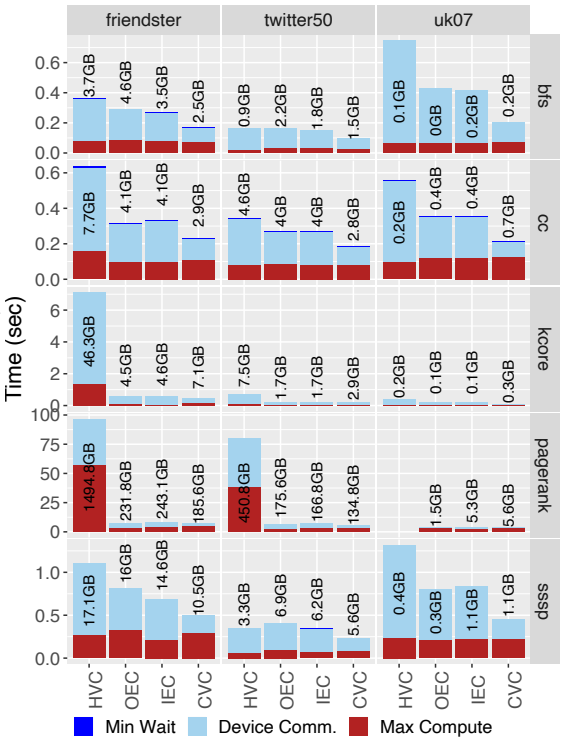

Figure 8: Breakdown of execution time of D-IrGL with different partitioning policies for medium graphs on 32 P100 GPUs of Bridges.

values for straggler hosts to compute with less stale values. However, Async may suffer from an increase in the redundant work performed. To illustrate, we examine bfs with uk14 on 64 GPUs. Async is slower due to increase in the minimum number of local rounds executed (from 1000 to 2141), leading to an increase in the average work items (from $6.7 * 10^{10}$ to $7.7 * 10^{10}$), communication volume, and host-device communication time. In contrast, for bfs with clueweb12 on 64 GPUs, the increase in redundant work is offset by a decrease in the minimum number of rounds executed by the straggler host, which improves overall execution time.

BASP-style execution is beneficial in most cases as it can reduce the idle time. However, it can lead more redundant computation/communication, so it would be useful to dynamically throttle the degree of asynchronous execution.

C. Multi-Host Multi-GPU Graph Analytics: Partitioning

In this subsection, we use all the optimizations — ALB + UO + Async (Var4), and analyze different partitioning policies.

Figure 7 shows the strong scaling of benchmarks with various partitioning policies for medium graphs. CVC scales best for all benchmarks and inputs. Figures 8 and 9 show the breakdown of execution time for medium and large graphs, respectively. The clear takeaway is that communication time is the bottleneck is most cases. The communication time of CVC is generally lower than all other partitioning policies even though it sends more data. Compute time may also be higher in some configurations, but in many cases the higher compute time is offset by the significantly lower communication time.

CVC has fewer communication partners due to the use of its structural invariants, thereby allowing it to send more data faster. This is similar to what has been previously observed for CPUs as well [7], except that CVC also helps reduce the host-device communication for GPUs. The key difference between the results on CPUs which used Sync and our results is that *CVC starts outperforming other policies at a much smaller scale — 16 or more GPUs*. This may be due to two reasons: (1) Async may balance the load better than Sync, or (2) the communication to computation ratio may be higher when GPUs are used instead of CPUs. This is important because none of the existing multi-GPU frameworks support vertex-cuts while hardware manufacturers are designing single-host multi-GPU systems with 16 GPUs (like NVIDIA DGX2).

Even with all the optimizations and the best partitioning policies, *the host-device communication time is a significant portion of the execution time* but not much more than the computation time. The execution time can be further reduced by overlapping this communication with computation using asynchronous communication between host and device or by communicating directly between devices using GPUDirect.

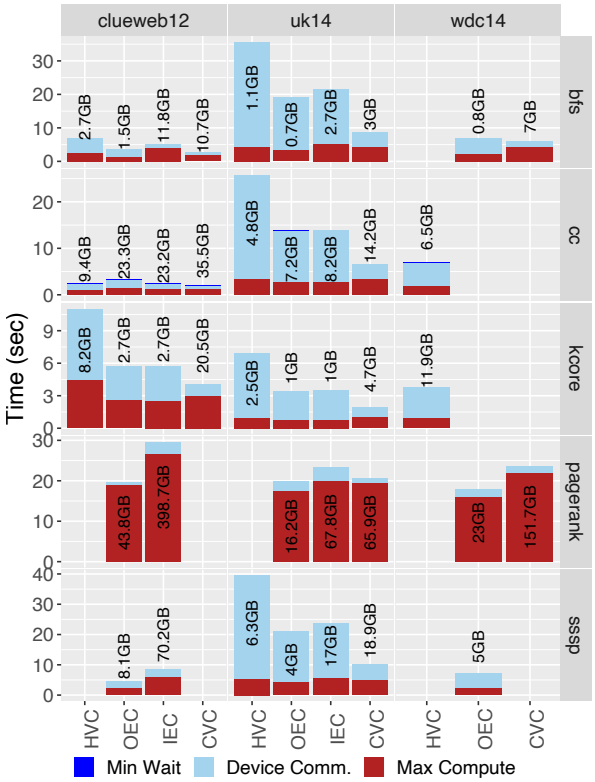

Figure 9: Breakdown of execution time of D-IrGL with different partitioning policies for large graphs on 64 P100 GPUs of Bridges.

Table IV shows the load balance for all benchmarks and policies with uk07 and uk14. Static load balance, dynamic load balance, and memory load balance refer to the distribution of edges, computation time, and GPU memory allocated, respectively. The metric for all three is the maximum of that measure divided by the mean of that measure. First, *static balance is not necessarily correlated to dynamic balance due to the irregular nature of graph analytics benchmarks*. For instance on uk14, pagerank with CVC is statically imbalanced but dynamically balanced, while bfs with IEC is statically balanced but dynamically imbalanced. Thus, even if a partitioning policy aims to statically balance load across GPUs, there is no guarantee that computation is balanced across GPUs. This is similar to what has been observed for CPUs [7]. Second, and more importantly, static and memory load balance are highly correlated as the amount of memory allocated on a GPU is proportional to the number of edges assigned to it. As GPUs have limited memory, static load balancing is important as *high static load imbalance may cause applications to run out of memory* even for graphs with size less than the combined memory available on all the GPUs. This is evident for partitioning policies on large graphs (Figure 9).

Table IV: Static load balance (max./mean no. of edges), dynamic load balance (max./mean compute time), and GPU memory (max./mean) of D-IrGL.

Benchmark	Partition	uk07 on 32 GPUs			uk14 on 64 GPUs		
		Static	Dynamic	Memory	Static	Dynamic	Memory
bfs	**CVC**	1.15	1.17	1.15	1.15	1.11	1.14
	HVC	1.10	1.20	1.08	1.40	1.38	1.38
	IEC	1.00	1.14	1.04	1.00	1.31	1.08
	OEC	1.00	1.20	1.02	1.00	1.24	1.03
cc	**CVC**	1.03	1.18	1.05	1.12	1.10	1.13
	HVC	1.09	1.30	1.08	1.11	1.34	1.11
	IEC	1.00	1.27	1.02	1.00	1.24	1.04
	OEC	1.00	1.29	1.02	1.00	1.22	1.04
kcore	**CVC**	1.03	1.14	1.05	1.12	1.12	1.13
	HVC	1.09	1.22	1.08	1.11	1.35	1.11
	IEC	1.00	1.20	1.02	1.00	1.31	1.04
	OEC	1.00	1.19	1.02	1.00	1.29	1.04
pagerank	**CVC**	1.16	1.04	1.15	1.15	1.02	1.14
	IEC	1.00	1.09	1.04	1.00	1.09	1.08
	OEC	1.00	1.10	1.03	1.00	1.08	1.04
sssp	**CVC**	1.15	1.10	1.15	1.15	1.09	1.15
	HVC	1.10	1.21	1.09	1.40	1.34	1.39
	IEC	1.00	1.14	1.02	1.00	1.24	1.04
	OEC	1.00	1.14	1.01	1.00	1.22	1.02

VI. Related Work

GPU Studies. Owens et al. [29] present a survey of general-purpose computation on GPUs. Shi et al. [14] present a survey of different issues in GPU-based graph processing, such as data layout, memory access pattern, and workload mapping. Gill et al. [7] present a detailed study of the different partitioning policies for distributed CPUs. Pan et al. [6] and Hoang et al. [30] analyze the performance of breadth-first search and triangle counting, respectively, on distributed GPUs. In this paper, we analyze five graph analytical applications on distributed GPUs with different partitioning policies as well computation and communication optimizations.

GPU Graph Analytical Frameworks. Several frameworks [31], [9], [27] focus on improving the performance of graph analytical applications on a single GPU by exploiting the architectural features. Totem [32] is a graph framework built to support Bulk Synchronous Parallel (BSP) execution on a CPU-GPU heterogeneous system. Gill et al. [33] propose a compiler framework for processing graph analytical applications on distributed CPU-GPU heterogeneous systems.

Medusa [34] provides a programming framework for writing graph analytical applications on single-host multi-GPU platforms by using sequential C/C++ code. Similarly, Gunrock [1] and Groute [2] provide support for writing single-host multi-GPU graph analytical applications using BSP and asynchronous style of execution respectively. D-IrGL [4] and Lux [3] support multi-host multi-GPU platforms. In this study, we evaluate distributed GPU graph analytical applications in D-IrGL [4] using real-world datasets of different sizes.

VII. Conclusion

Our study on distributed multi-GPU graph analytical system shows the need for vertex-cut policies such as Cartesian vertex-cut for scaling on multi-GPU platforms. It also shows the importance of static balance of graph partitions to efficiently handle large graphs using the limited GPU memories.

To improve performance on multi-host multi-GPU systems, frameworks should adopt modern GPU architecture capabilities such as GPUDirect to avoid data transfers through the host. In addition, control mechanisms need to be developed to dynamically throttle bulk-asynchronous execution to obtain the right trade-off between decoupled execution of hosts and redundant computation/communication.

Acknowledgment

This research was supported by the NSF grants 1406355, 1618425, 1705092, 1725322, and by the DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563. We used XSEDE grant ACI-1548562 through allocation TG-CIE170005. We used the Bridges cluster, supported by NSF award number ACI-1445606.

References

[1] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens, "Multi-GPU Graph Analytics," in *IPDPS*, 2017.

[2] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations," in *PPoPP*, 2017.

[3] Z. Jia, Y. Kwon, G. Shipman, P. McCormick, M. Erez, and A. Aiken, "A Distributed multi-GPU System for Fast Graph Processing," *VLDB*, 2017.

[4] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A Communication-optimizing Substrate for Distributed Heterogeneous Graph Analytics," in *PLDI*, 2018.

[5] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, V. Jatala, V. K. Nandivada, M. Snir, and K. Pingali, "Gluon-Async: A Bulk-Asynchronous System for Distributed and Heterogeneous Graph Analytics," in *PACT*, 2019.

[6] Y. Pan, R. Pearce, and J. D. Owens, "Scalable Breadth-First Search on a GPU Cluster," in *IPDPS*, 2018.

[7] G. Gill, R. Dathathri, L. Hoang, and K. Pingali, "A Study of Partitioning Policies for Graph Analytics on Large-scale Distributed Platforms," ser. PVLDB, 2018.

[8] L. Hoang, R. Dathathri, G. Gill, and K. Pingali, "CuSP: A Customizable Streaming Edge Partitioner for Distributed Graph Analytics," in *IPDPS*, 2019.

[9] S. Pai and K. Pingali, "A Compiler for Throughput Optimization of Graph Algorithms on GPUs," in *OOPSLA*, 2016.

[10] R. Chen, J. Shi, Y. Chen, and H. Chen, "PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs," in *EuroSys*, 2015.

[11] E. G. Boman, K. D. Devine, and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2D graph partitioning," in *SC 2013*.

[12] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The TAO of parallelism in algorithms," in *PLDI*, 2011.

[13] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri, "Partitioning Trillion-Edge Graphs in Minutes," in *IPDPS*, 2017.

[14] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua, "Graph Processing on GPUs: A Survey," *ACM Comput. Surv.*, 2018.

[15] R. Nasre, M. Burtscher, and K. Pingali, "Data-driven versus Topology-driven Irregular Computations on GPUs," in *IPDPS*, 2013.

[16] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU Graph Traversal," *SIGPLAN Not.*, 2012.

[17] V. Jatala, L. Hoang, R. Dathathri, G. Gill, V. K. Nandivada, and K. Pingali, "An Adaptive Load Balancer For Graph Analytical Applications on GPUs." [Online]. Available: http://arxiv.org/abs/1911.09135

[18] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr, "Xsede: Accelerating scientific discovery," *Computing in Science and Engineering*, 2014.

[19] D. Chakrabarti, Y. Zhan, and C. Faloutsos, *R-MAT: A Recursive Model for Graph Mining*. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/1.9781611972740.43

[20] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[21] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered Label Propagation: A Multiresolution Coordinate-free Ordering for Compressing Social Networks," in *WWW*, 2011.

[22] P. Boldi, A. Marino, M. Santini, and S. Vigna, "Bubing: Massive crawling for the masses," in *WWW*, 2014.

[23] P. Boldi and S. Vigna, "The WebGraph framework i: Compression techniques," in *WWW*, 2004.

[24] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer, "Web data commons - hyperlink graphs," 2012. [Online]. Available: http://webdatacommons.org/hyperlinkgraph/

[25] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing Locality and Independence with Logical Regions," in *SC 2012*.

[26] D. Bonachea, "GASNet Specification, V1.1," Tech. Rep., 2002.

[27] 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 *PPoPP*, 2015.

[28] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM J. Sci. Comput.*, 1998.

[29] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell, "A Survey of GeneralPurpose Computation on Graphics Hardware," *Computer Graphics Forum*, 2007.

[30] L. Hoang, V. Jatala, X. Chen, U. Agarwal, R. Dathathri, G. Gill, and K. Pingali, "DistTC: High Performance Distributed Triangle Counting," in *HPEC Graph Challenge '19*.

[31] K. Meng, J. Li, G. Tan, and N. Sun, "A Pattern Based Algorithmic Autotuner for Graph Processing on GPUs," in *PPoPP*, 2019.

[32] A. Gharaibeh, L. Beltrão Costa, E. Santos-Neto, and M. Ripeanu, "A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing," in *PACT*, 2012.

[33] G. Gill, R. Dathathri, L. Hoang, A. Lenharth, and K. Pingali, "Abelian: A Compiler for Graph Analytics on Distributed, Heterogeneous Platforms," in *Euro-Par*, 2018.

[34] J. Zhong and B. He, "Medusa: Simplified Graph Processing on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, 2014.