GX-Plug: a Middleware for Plugging Accelerators to Distributed Graph Processing

Kai Zou, Xike Xie*, Qi Li, Deyu Kong
University of Science and Technology of China
{slnt, likamo, cavegf}@mail.ustc.edu.cn, *xkxie@ustc.edu.cn

***Abstract*—Recently, research communities highlight the necessity of formulating a scalability continuum for large-scale graph processing, which gains the scale-out benefits from distributed graph systems, and the scale-up benefits from high-performance accelerators. To this end, we propose a middleware, called GX-plug, for the ease of integrating the merits of both. As a middleware, GX-plug is versatile in supporting different runtime environments, computation models, and programming models. More, for improving the middleware performance, we study a series of techniques, including pipeline shuffle, synchronization caching and skipping, and workload balancing, for intra-, inter-, and beyond-iteration optimizations, respectively. Experiments show that our middleware efficiently plugs representative distributed graph systems, e.g., GraphX and Powergraph, to accelerators, with up-to 20x acceleration ratio.**

***Index Terms*—Distributed graph systems, Middleware, accelerators**

I. Introduction

Big graph analytics are often with large data volumes, high computation intensiveness, and diversified applications, such as social networks, Internet connectivity analysis, traffic networks, and biological structures, just to name a few. To meet the scaling-out challenge [1], an increasing number of distributed graph systems, including GraphX [2], PowerGraph [3], are proposed and deployed. To meet the scaling-up challenge, non-distributed graph systems incorporate emerging accelerators, such as Gunrock [4] with GPU and ThunderGP [5] with FPGA. Beyond merely scale-out or scale-up, recent research spotlights the vision of a scalability continuum [6], where distributed graph systems and accelerators can be integrated for elastic scaling of big graph systems deployed in data centers.

The high computational concentration in data centers makes an appealing case for accelerating distributed graph processing. For example, data centers provide a range of accelerator instances [7], e.g, GPU and CPU, for flexibly scaling up the performance to application demands. For non-graph systems, the practicability of integrating scaling-up and -out solutions has been recognized. For example, Nvidia has announced the plan to support Spark 3.0 with GPU acceleration in 2020 [8], [9]. For graph systems, it is more challenging, because there exist a large number of system variants [6], due to the diversity and irregularity of distributed graph processing in data centers. They are with different architectures, runtime environments (Java and C++), and computation models[1]. In this work, we propose a middleware, GX-Plug, for big graph processing, where accelerators can be neatly plugged to heterogenous distributed graph systems deployed in data centers.

Our middleware follows an agile framework, called *daemon-and-agent*. Daemon enables transparency of computation accelerators, e.g., GPU and CPU, to upper (distributed graph) systems, e.g., GraphX and PowerGraph, by holding customizable graph programming interfaces. Agent is for bridging upper systems and daemons, covering data exchanging, graph algorithm interface invoking, and daemon life-cycle controlling. With the daemon-agent framework, the middleware shows flexibility in supporting different runtime environments, computation models, and programming models.

For easy accessing to accelerators, daemon provides a graph algorithm template, based on conventional iterative models, and support transplanting existing distributed graph algorithms with ease. Agent provides a kit of interfaces to cooperate with daemons and uppers systems for global graph computation. Accordingly, it takes only a few lines of code to plug accelerators to upper systems.

Nevertheless, there arise a series of research challenges for implementing the middleware, besides the software design and development efforts on the adaption to different runtimes (Java and C++) and different accelerators (GPU and CPU). First, there exists considerable data transmission overhead for the middleware in delivering and translating the data payloads into desired formats, between upper systems and accelerators, causing *intra-iteration* overhead. Second, the irregular and complex graph structure incurs imbalanced workloads, as well as latencies in frequent global synchronization, causing *inter-iteration* overhead. Third, it is difficult to schedule the workload and computation resource for different tasks and system configurations, recognized as *beyond-iteration* overhead. The overheads can much degrade the middleware performance.

In this work, we tackle the first challenge by incorporating pipeline shuffle for optimizing the data transferring between daemons and agents. We tackle the second challenge by optimizing the process of data synchronization, including

[1]BSP (Bulk Synchronous Parallel) is a parallel model that performs computation in iterative steps, including three steps of computation, communication, and synchronization. BSP model has been the most fundamental and popular execution approach on distributed graph systems. GAS (Gather-Apply-Scatter) model is another basic and widely adopted model for distributed graph processing, based on BSP [10]. (BSP [11], GAS [3], etc.), and programming models (vertex-centric, edge-centric, etc.)

caching and skipping, so as to minimize the volumes of data transferring during the synchronization phase. We tackle the third challenge by making the size of transferred data blocks self-adaptive to the workloads of distribute nodes, and therefore the system workload balancing can be improved.

In this work, we focus on the implementation and optimization of the middleware. We are aware of techniques on graph processing, either on the accelerator end, e.g., exploring memory hierarchies [12], [13] for accelerating on-chip data accessing and reinforcing local GPU processing networks with NVLink and NVSwitch [14], [15]; or on the upper system end, e.g., using RDMA [16] for faster distributed system communication and using pull-push model [17] for data transferring optimization in specific applications. We would like to argue that optimizations merely on upper system or accelerator end are beyond the scope of the middleware, and are orthogonal to our work.

Our contributions can be listed as follows.

- We propose, to our best knowledge, the first middleware for arming distributed graph systems with high-performance accelerators, to meet the scale-out and -up needs of large-scale graph analytics.
- For the middleware, we design a novel daemon-agent framework, which achieves flexible deployment on different upper systems and easy accessing to accelerators.
- The middleware is general in supporting different computation models, such as BSP and GAS. Existing distributed graph algorithms can be transplanted for accessing accelerators with ease.
- For the middleware optimization, we investigate a series of techniques, such as pipeline shuffling, synchronization caching and skipping, and workload balancing, for intra-, inter-, and beyond-iteration optimizations, respectively.
- We conduct extensive experiments on real datasets to evaluate the efficiency and scalability of the middleware.

The rest of the paper is organized as follows. Section II shows the overview of the daemon-agent framework of the middleware. Section III investigates optimization techniques used to improve internal performance of middleware. Section IV discusses middleware deployment technique in real world. Section V reports the results of empirical studies. Section VI presents related works. Section VII concludes the paper.

II. Middleware Overview

Cloud services are witnessed to evolve from cloud storage services, comprised of a multitude of distributed nodes/machines/instances, to high-performance cloud computing services, comprised of accelerator-powered distributed nodes. For example, accelerators, like GPUs and multi-core CPUs, etc., are being deployed to distributed nodes in top cloud services, such as Amazon EC2, Google cloud, and Microsoft Azure Blob. Our middleware is to boost graph computing on such cloud services, supporting system configuration and application development with ease. A bird's eye view of the middleware is shown in Figure 1.

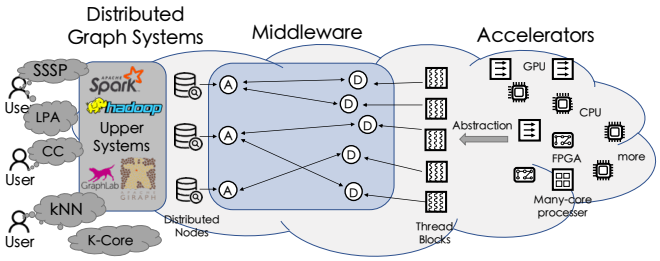


Fig. 1: Middleware Overview (A: Agent, D: Daemon)

Our middleware follows an agile daemon-agent framework. A daemon is combined with a *threadblock*, an abstract representation of accelerators. A agent resides in a distributed node, for the connection with upper systems. A agent connects one or more daemons, the number of which can be tuned to control the number of accelerators allocated, for flexible computation power distribution and workload balancing.

In the sequel, we investigate the daemon-agent framework, which is the core of the middleware, in Section II-A. We also study the data storage and controllers of the middleware, in Sections II-B and II-C.

A. Daemon-Agent Framework

The structures of daemon and agent, and their interactions are shown in Figure 2. In general, daemons are in connection with accelerators, and agents are in connection with upper systems. The communication of the two parts is done through System V IPC for efficiency.

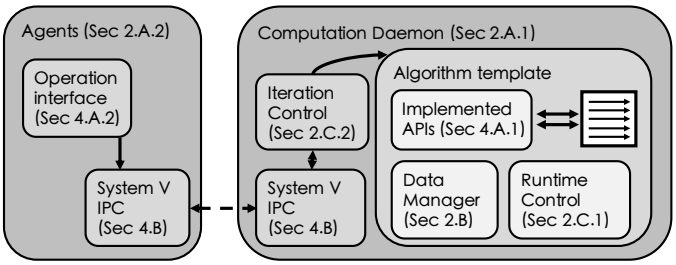


Fig. 2: Daemon-Agent Framework

1) Daemon: Daemon is the part where graph algorithms execute. A daemon holds an algorithm template and the iteration logic controlling, as shown in Figure 2. The design of daemon is towards transparent hardware management to upper systems. For tuning computation powers, system developers can control the number of daemons. For accelerating distributed graph algorithms, algorithm engineers only need to focus on the implementation of template APIs.

At the runtime, the system implements an instance of the pre-designed algorithm template for daemons. The connection with accelerators are established during the initialization phase, and details are hidden to system developers since then.

2) Agent: Agents are designed as a bridge for upper systems and daemons. The structure of an agent is shown in Figure 2. Essentially, an agent is designed as a set of

operation interfaces between upper systems and daemons, on data exchanging, subfunction execution, and daemon lifecycle controlling. With the operation interfaces on agents, upper systems can substantially control daemons, getting rid of the burden on compatibility of software and hardware environments.

In the local environment of a distributed node, there should be at least one agent, and there can be multiple daemons, representing different accelerators. Also, as shown in Section III, agents are equipped with a series of optimization techniques to reduce the overhead caused by data transferring, which is observed as the major source that affects system performance.

B. Data Storage

General purpose graph computation involves operations on attributes of vertices and corresponding edges, which are related to data storage. There are two mainstream ways to store graphs, called vertex-centric and edge-centric storage strategies.

Our middleware is general in supporting either vertex- or edge-centric storage strategy. Figure 3 shows the overview of graph data storage of the middleware.

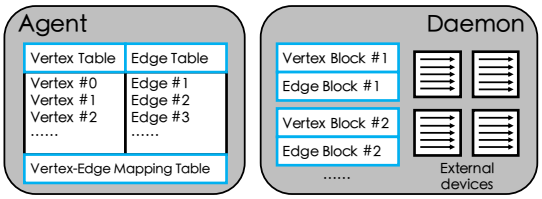


Fig. 3: Data Storage Overview

For the daemon part, edge-centric storage is adopted, because it is shown to be more effective in workload balancing, than its counterpart [18], [19], especially for real graphs following power-law distributions [20]. Daemon maintains a series of data blocks to be loaded for accelerators to process, including vertex blocks and edge blocks. Each edge block contains a fixed number of edges. Each edge block is also associated with a paired vertex block. For each edge of the edge block, either its source or destination vertex can be found in its paired vertex block.

Of a distributed node, the block size is related to workload balancing, parallism maximization, and therefore runtime efficiency. It can be tuned in accordance to the computation power connected with the daemon. More optimization details are covered in the next section.

For the agent part, graph data are stored in a vertex table, an edge table, and a mapping table, which maps a vertex in the vertex table to a set of its outer edges in the edge table. The purpose of the intermediate data storage is to: 1) be general for both edge- and vertex-centric storage, and reduce storage redundancies; 2) be efficient in retrieving and modifying graph elements (a vertex or an edge) if necessary; 3) support synchronization caching mechanism mentioned in Section III-B as auxiliary data structures.

To construct an edge block, agent selects a vertex and puts its outer edges to the current edge block, with vertex-edge mapping table. Also, destination vertices of these edges and their attributes are put into corresponding vertex block. The middleware repeatedly selects vertices that need for computation, until the current edge block is fully packed. The selection stops until all vertices that need for computation are checked. In computation, the data blocks are accessed by techniques mentioned in Section III-A, for the data transferring optimization, between daemons and agents.

C. Controllers

For the middleware, daemon is in charge of orchestrating different computational components. We introduce two components located in the daemon, making the middleware adaptable to different computation models and optimize the iterative processing for upper systems.

1) Runtime Control: The runtime control component is for controlling the execution order of implemented APIs, including the runtime information collected from accelerators and sending /receiving flags for iteration controlling. By controlling the execution order of implemented APIs, the middleware can easily be integrated into different computation models, as discussed in Section IV-C2.

2) Iteration Control: The iteration control component is for controlling and coordinating the whole iteration, including the computation. Since the middleware separates the runtimes of upper systems and computation, it is necessary to connect both parties for in-between data synchronization and the computation processing cycle. Several optimizations, such as pipeline shuffling, synchronization caching and skipping, are implemented in Section III. The component collaboratively works with agents for retrieving information from upper systems, and works with the runtime control component for exchanging flag information, to fulfill the iteration controlling.

The purpose of the middleware is for easy acceleration of distributed graph processing. However, the middleware itself may bring in additional overhead, which may potentially degrade the system performance. In the sequel, we devise a series of optimizations to alleviate or even eliminate the overhead originating from the middleware.

III. Runtime Optimization

In this section, we introduce three optimization techniques, pipeline shuffling for improving intra-iteration processing, synchronization caching and skipping for eliminating unnecessary data transferring for inter-iteration processing, and workload balancing for beyond-iteration optimization.

A. Intra-Iteration Optimization: Pipeline Shuffling

1) Motivation: For the basic daemon-agent framework aforementioned, an ordinary workflow of graph processing acceleration consists of five steps, data downloading (from upper systems), agent-to-daemon data transferring, computing, daemon-to-agent data transferring, and data uploading (to upper systems). However, a tightly coupled execution of the

five steps, where the output of one step is streamed as the input of another step, leads to many waiting-and-suspending states and therefore the underutilization of computation resources.

For example, the computing step must wait for agent-to-daemon data transferring to start, so that the computing step would be suspended during other steps. To alleviate the predicament and to improve the computation resource utilization, we investigate a pipeline parallelism mechanism, *Pipeline shuffle*, to the middleware.

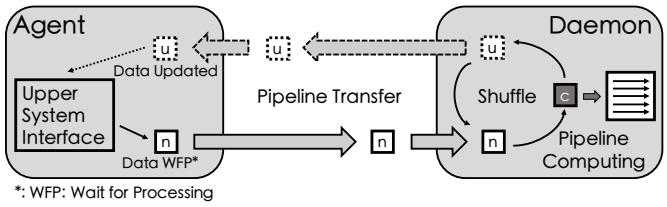


Fig. 4: Pipeline Shuffle

Algorithm 1 Pipeline Shuffle - Daemon Side

Input: Computer Device com_dev, Data area pointer n, c, u

```
1: while In Iteration do
2:     Block_Recv(agent, msg)
3:     if msg = "ExchangeFinished" then
4:         Rotate(n → c → u → n)
5:         Send(agent, "RotateFinished")
6:     else if c contains contents to compute then
7:         com_dev.Load(*c)
8:         com_dev.Compute()
9:         *c ← com_dev.data
10:        Send(agent, "ComputeFinished")
11:    else
12:        Send(agent, "ComputeAllFinished")
13:        End_Iteration()
```

Algorithm 2 Pipeline Shuffle - Agent Side

Input: Upper system interface USI, Data area pointer n, c, u

```
1: *n ← USI.Download()
2: Send(daemon, "ExchangeFinished")
3: while In Iteration do
4:     Block_Recv(daemon, msg)
5:     if msg = "RotateFinished" then
6:         upload, download = new Thread()
7:         for Thread upload do
8:             USI.Upload(*u)
9:         for Thread download do
10:            *n ← USI.Download()
11:    else if msg = "ComputeFinished" then
12:        if upload.isTerminated() then
13:            if download.isTerminated() then
14:                Send(daemon, "ExchangeFinished")
15:    else if msg = "ComputeAllFinished" then
16:        if upload.isTerminated() then
17:            if download.isTerminated() then
18:                End_Iteration()
```

2) Overview: The idea is to construct a multi-layer pipeline for better parallelism. First, we replace the original 5-step data transferring into a 3-step data transferring, data downloading, computing, and data uploading, which are handled by 3 threads, Thread.Download, Thread.Compute and Thread.Upload, respectively. Compared to 5-step setting, the 3-step data transferring eliminates the two steps of agent-to-daemon and daemon-to-agent data transferring. We show how the elimination is exactly done by using system V IPC in Section IV-B. Second, based on the 3-step setting, we construct a 3-layer pipeline to reduce the suspension time of the computing step, and finally improve computation resource utilization. The overview of pipeline shuffle is shown in Figure 4. The detailed process is depicted by Algorithms 1 and 2.

a) Pipeline Parallelism: The pipeline consists of 3 layers in correspondence to the 3 steps mentioned above, as shown in Figure 5. With the pipeline, an iteration can be decomposed into a sequence of 3 processing cycles, corresponding to the 3 layers. For all three pipeline layers, we use "edge triplets" as the intermediate data structure, which includes an edge and its source and destination vertices, by efficiently joining the edge and vertex tables. With the data structure of edge triplets, the pipeline has homogenous data structures for all layers, for avoiding unnecessary data format transformation and enabling grained-granularity data retrieval. Essentially, the triplet is the basic processing unit of an iteration, which serves as both the source of computation input and the carrier of output. Within an iteration, there is no data dependencies between triplets.

For each layer, triplets are grouped into a set of blocks, as shown in Figure 5. The blocks are assigned to the 3 threads for processing. Thus, pipeline parallelism can be established which significantly improves the system performance.

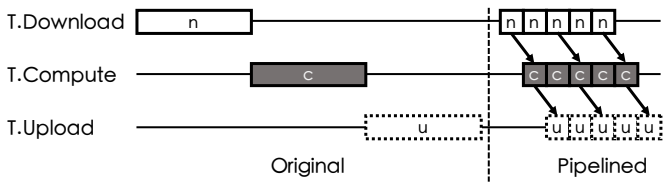


Fig. 5: Pipelined Process Flow with Pipeline Shuffle

b) Shuffle for data transfer reduction: There are intermediate data layers existing in an ordinary pipeline design, as shown in Figure 5. The i-th data layer stores the result of i-th pipeline process cycle, and is used for $(i+1)$-th downstream pipeline process cycle. Within a cycle, data are transferred between threads for fulfilling the processing. However, frequent data copying incurs considerable system overhead.

To handle that, we design a shuffle mechanism for efficient data transferring in pipeline parallelism. First, we allocate equal sized memory chunks for threads. Each memory chunk is associated with a pointer, for the reference of the front block currently being processed inside the chunk. Second, inter-thread data copying is replaced by the pointer copying. For example, in Figure 5, data blocks of 3 layers can be represented by n-block, c-block, and u-block, indicating blocks for

new data retrieved from upper systems, blocks for computing, and blocks for uploading to upper systems, respectively. When a pipeline cycle is finished, the 3 pointers are shuffled in a rotation manner: the pointer to n-block switches to c-block, the pointer to c-block switches to u-block, and so on. With the pipeline shuffle mechanism, there is no more data copying between threads, since it is completed in situ.

c) Block Size Selection: In our work, we found the size of a block has a profound effect over the parallelism performance. We assume that there is a sub-dataset distributed to a agent-daemon pair for processing which contains d entities need to be processed in the current iteration. Also, agent divides the dataset into s blocks evenly, $b = \frac{d}{s}$. Let $T_n(b)$, $T_c(b)$, $T_u(b)$ be the process time of one data block in Thread.Download, Thread.Compute and Thread.Upload, respectively. T_n and T_u are proportional to the data block size. We can estimate pipeline processing time T_{total}.

$$\begin{aligned} T_{total} &= T_n(b) + \max(T_n(b), T_c(b)) \\ &+ (s-2)\max(T_n(b), T_c(b), T_u(b)) \\ &+ \max(T_c(b), T_u(b)) + T_u(b) \end{aligned} \tag{1}$$

T_c refers to the time cost of Thread.Compute, and consists of calling computation devices, loading data to devices, and computation. Their corresponding time costs are represented by T_{call}, T_{comp}, and T_{copy}, respectively. The operation of calling devices takes constant time, while computation and data copying time are related to data size. So, $T_c(b)$ can be modeled as follows.

$$T_c(b) = T_{call} + T_{comp}(b) + T_{copy}(b)$$

When s increases, block size b decreases, so do T_n and T_u. T_c also decreases when b decreases, but will never be less than T_{call}, which means T_{total} starts increasing when s is large enough and keep increasing. On the other hand, T_n and T_u increase when s is being smaller, since b is being larger. Thus, both the function $T_{total}(b)$ and $T_{total}(s)$ should tend to become a U-turn form. Thus, s and b should be deliberately configured for achieving fine-tuned system performance. We will try to calculate the value of b in order to provide optimization suggestion to overall system.

The calculation follows two assumptions: 1) The sub-dataset distributed to the agent-daemon pair has d entities, and is divided into s blocks evenly, which have the size $b = \frac{d}{s}$. 2) we assume both T_n, T_u are directly proportional to the block size; and 3) In T_c shown in Equation 2, T_{call} has a fixed number, while T_{comp} and T_{copy} are also directly proportional to b. We use k_1, k_2 and k_3 to represent download, computation, and upload time cost per unit of data entity. Thus, Equation 1 can be further simplified as follows:

$$\begin{aligned} T_{total} &= k_1 b + \max(k_1 b, a + k_2 b) \\ &+ (s-2)\max(k_1 b, a + k_2 b, k_3 b) \\ &+ \max(a + k_2 b, k_3 b) + k_3 b \end{aligned} \tag{2}$$

Based on the problem description, Lemma 1 shows the derivation of optimal b for the pipeline shuffle mechanism, as follows.

Lemma 1. *If a distributed node stores d data entities, the optimal block size b_{opt} and corresponding $T_{total_{min}}$ can be calculated as follows.*

$$b_{opt} = \begin{cases} \frac{a}{k_1 - k_2}, & \left(\begin{array}{c} k_{max} = k_1, \& \\ \frac{a}{k_1 - k_2} < Q \end{array}\right) \\ \frac{a}{k_3 - k_2}, & \left(\begin{array}{c} k_{max} = k_3, \& \\ \frac{a}{k_3 - k_2} < Q \end{array}\right) \\ \sqrt{\frac{ad}{k_1 + k_3}}, & otherwise \end{cases} \tag{3}$$

, and

$$T_{total_{min}} = \begin{cases} \frac{a(k_1 + k_3)}{k_1 - k_2} + k_1 d, & \left(\begin{array}{c} k_{max} = k_1, \& \\ \frac{a}{k_1 - k_2} < Q \end{array}\right) \\ \frac{a(k_1 + k_3)}{k_3 - k_2} + k_3 d, & \left(\begin{array}{c} k_{max} = k_3, \& \\ \frac{a}{k_3 - k_2} < Q \end{array}\right) \\ k_2 d + 2\sqrt{(k_1 + k_3)ad}, & otherwise \end{cases} \tag{4}$$

, where $Q = \sqrt{\frac{ad}{k_1 + k_3}}$.

Proof. Following Equation 2, we have 3 cases to consider.

Case 1: $T_n = k_1 b$ is the maximum value. This case is true only if k_1 is the maximum of the 3 parameters, k_1, k_2, and k_3. Accordingly, b should satisfy the follows.

$$k_1 b \geq a + k_2 b \Rightarrow b \geq \frac{a}{k_1 - k_2}$$

Thus, Equation 2 can be transformed into:

$$\begin{aligned} T_{total} &= s k_1 b + \max(a + k_2 b, k_3 b) + k_3 b \\ &= k_1 d + \max(a + k_2 b, k_3 b) + k_3 b \end{aligned}$$

Notice that a and $\{k_i\}_{i \leq 3}$ are all positive, and both $\max(a + k_2 b, k_3 b)$ and $k_3 b$ increase when b increases. Thus, when $b = \frac{a}{k_1 - k_2}$, we have the minimum value of T_{total} as follows.

$$\begin{aligned} T_{total} &= k_1 d + \max(a + \frac{a k_2}{k_1 - k_2}, \frac{a k_3}{k_1 - k_2}) + \frac{a k_3}{k_1 - k_2} \\ &= k_1 d + \max(\frac{a k_1}{k_1 - k_2}, \frac{a k_3}{k_1 - k_2}) + \frac{a k_3}{k_1 - k_2} \\ &= k_1 d + \frac{(k_1 + k_3)a}{k_1 - k_2} \end{aligned}$$

Case 2: $T_c = (a + k_2 b)$ is the maximum value. First, we have this equation below, where $s = \frac{d}{b}$.

$$\begin{aligned} T_{total} &= k_1 b + s(a + k_2 b) + k_3 b \\ &= (k_1 + k_3)b + k_2 d + \frac{ad}{b} \end{aligned}$$

In this equation, we can have the minimum T_{total}, if b equals $\sqrt{\frac{ad}{k_1 + k_3}}$, which is Q. Notice that b may not equal Q, as constrained by $\{k_i\}$. Accordingly, we discuss T_{total} in 3 subcases, based on the value of k_2.

(k_2 is the minimum one.) In this situation, both $(k_1 - k_2)$ and $(k_3 - k_2)$ are positive. Thus, we have:

$$a + k_2 b \geq \max(k_1, k_3) \cdot b \Rightarrow b \leq \min(\frac{a}{k_1 - k_2}, \frac{a}{k_3 - k_2})$$

Assume that $k_1 \geq k_3$, we have $b \leq \frac{a}{k_1-k_2}$. Thus, we have the minimum T_{total}:

$$T_{total} = \begin{cases} k_2 d + 2\sqrt{(k_1+k_3)ad}, & \frac{a}{k_1-k_2} \geq Q \\ \frac{a(k_1+k_3)}{k_1-k_2} + k_2 d + (k_1-k_2)d, & \frac{a}{k_1-k_2} < Q \end{cases}$$

Also, we have the minimum T_{total} when $k_3 \geq k_1$:

$$T_{total} = \begin{cases} k_2 d + 2\sqrt{(k_1+k_3)ad}, & \frac{a}{k_3-k_2} \geq Q \\ \frac{a(k_1+k_3)}{k_3-k_2} + k_2 d + (k_3-k_2)d, & \frac{a}{k_3-k_2} < Q \end{cases}$$

(k_2 is the middle one.) In this situation, we should notice the change of the inequality, because some terms of Equation 5 can be negative. Without loosing generality, we assume $k_3 \leq k_2 \leq k_1$. In this case, $(k_1 - k_2)$ is positive, and $(k_3 - k_2)$ is negative. Thus, we can have $b \leq \frac{a}{k_1-k_2}$, since

$$\frac{a}{k_3-k_2} < 0 \leq b \leq \frac{a}{k_1-k_2} \Rightarrow 0 \leq b \leq \frac{a}{k_1-k_2} \quad (5)$$

Then, we have the minimum value of T_{total} as shown in Equation 5.

$$T_{total} = \begin{cases} \frac{a(k_1+k_3)}{k_1-k_2} + k_2 d + (k_1-k_2)d, & \frac{a}{k_1-k_2} < Q \\ k_2 d + 2\sqrt{(k_1+k_3)ad}, & otherwise \end{cases}$$

On the other hand, if $k_3 \geq k_2 \geq k_1$ holds, the minimum value of T_{total} is:

$$T_{total} = \begin{cases} \frac{a(k_1+k_3)}{k_3-k_2} + k_2 d + (k_3-k_2)d, & \frac{a}{k_3-k_2} < Q \\ k_2 d + 2\sqrt{(k_1+k_3)ad}, & otherwise \end{cases}$$

(k_2 is the maximum one.) In this situation, both $k_1 - k_2$ and $k_3 - k_2$ are negative. Since $b > 0$, b is also greater than $\frac{a}{k_1-k_2}$ and $\frac{a}{k_3-k_2}$. Thus, we have the minimum T_{total} when $b = \sqrt{\frac{ad}{k_1+k_3}} = Q$:

$$T_{total} = k_2 d + 2\sqrt{(k_1+k_3)ad}$$

Case 3: $T_u = k_3 b$ is the maximum value. This case is true only if k_3 is the maximum of the 3 parameters in k_1, k_2, and k_3.

Following the discussion in Equation 5, we simply have the conclusion that T_{total} has the minimum value $k_3 d + \frac{(k_1+k_3)a}{k_3-k_2}$ when $b = \frac{a}{k_3-k_2}$.

Discussion: Following the previous discussion, and the order of k_1, k_2 and k_3, we have 3 cases to calculate b_{opt}.

Case (i). k_1 is the maximum one: if $\frac{a}{k_1-k_2} \geq \sqrt{\frac{ad}{k_1+k_3}} = Q$, $b = \sqrt{\frac{ad}{k_1+k_3}} = Q$, and T_{total} have the minimum value $k_2 d + 2\sqrt{(k_1+k_3)ad}$. Otherwise, $b = \frac{a}{k_1-k_2}$, and T_{total} have the minimum value $k_1 d + \frac{(k_1+k_3)a}{k_1-k_2}$.

Case (ii). k_2 is the maximum one: $b = \sqrt{\frac{ad}{k_1+k_3}} = Q$, and T_{total} have the minimum value $k_2 d + 2\sqrt{(k_1+k_3)ad}$.

Case (iii). k_3 is the maximum one: if $\frac{a}{k_3-k_2} \geq \sqrt{\frac{ad}{k_1+k_3}} = Q$, $b = \sqrt{\frac{ad}{k_1+k_3}} = Q$, and T_{total} have the minimum value $k_2 d + 2\sqrt{(k_1+k_3)ad}$. Otherwise, $b = \frac{a}{k_3-k_2}$, and T_{total} have the minimum value $k_1 d + \frac{(k_1+k_3)a}{k_3-k_2}$.

Then, Equations in Lemma 1 can be directly derived from the above discussion, by grouping b and k. □

Notice that both s and b must be integers. If b_{opt} or $s_{opt} = \frac{d}{b_{opt}}$ is not an integer, we choose 2 values $\lfloor s_{opt} \rfloor$ and $\lceil s_{opt} \rceil$ for s, and 2 values $\lfloor b_{opt} \rfloor$ and $\lceil b_{opt} \rceil$ for b, so that Equation 2 can be used for estimating the minimum T_{total}.

B. Inter-Iteration Optimization: Synchronization Caching and Skipping

1) Motivation: For a distributed graph system, there is inevitable data synchronization between iterations, for ensuring the data correctness in every distributed node. However, it is often costly to do such data synchronization, since it would trigger considerable data copying between two successive iterations. Also, in a naïve integrated scale-out and -up system, the data copying involves memory accessing from distributed system environments to external computation accelerators, which is costly. Thus, it is necessary to reduce the load of data synchronization, either for the number of times that synchronization triggers, or for the data volume transferred. To this end, we introduce techniques, called *synchronization caching* and *synchronization skipping*, for inter-iteration optimization.

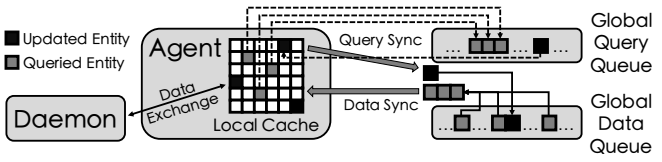


Fig. 6: Synchronization Caching

2) Synchronization Caching: Figure 6 shows the main process of synchronization caching. The idea is to use local cache of agents to reduce unnecessary data transferring between daemons and upper systems, including two parts, *LRU-based caching* and *lazy uploading*.

a) LRU-based Caching: First, we show the observation of redundant data transferring. Think twice of the data transferring process. At the beginning of each iteration, agent downloads data entities to be used by daemon computation, from upper systems. If a vertex is involved in several iterations of computation, but without attributes being updated, the corresponding information still has to be downloaded from uppers systems for multiple times.

To solve this problem, agent caches a set of vertices in a temporary vertex table, and the cache is organized in a least recently used (LRU *in short*) manner. Initially, when entering the cache, every vertex has a weight, whose value decreases with the passage of iterations, and increases if being used for computation. When daemon requires a specific vertex for computation, agent first checks its local cache for the vertex.

If not found, agent downloads it from upper systems to cache, and evicts the vertex with the highest weight. When agent collects computation results for updating to upper systems, it first checks if corresponding vertices are cached. If so, agent updates the attributes of the vertex, and upgrades its weight. Otherwise, agent chooses vertices with the lowest weights, and replaces them by vertices in the computation result. If the chosen vertices have been updated in previous iterations, corresponding information will be uploaded to upper systems. The updated vertices in the cache are marked, for lazy uploading, as discussed below.

b) Lazy Uploading: Also, there is no need for immediate uploading an updated vertex, until it is involved in computation happened in other distributed nodes. For example, if there are many copies of a vertex generated before synchronization, only the vertex copy with optimal updated value needs to be uploaded, which means other vertex copies are no longer needed for both data uploading and synchronization. To prevent unnecessary immediate uploading, we make the uploading being "lazy".

So, we design two queues for the caching mechanism, *global query queue* and *global data queue*. After all computation results are updated to the cache, agent first constructs a list of vertex IDs which are needed by the local node for the next iteration. All agents then push their local lists to the upper system, the union of which formulates the global query queue. This global query queue is broadcast to all agents. Every agent compares its cache with the global query queue, and uploads the required vertices to the global data queue. This way, data transferring is triggered only if it is necessary. Algorithm 3 shows the details of lazy uploading.

Algorithm 3 Lazy Uploading

Input: Updated Dataset s, Global query queue gqq, Global data queue gdq

1: $s_q \leftarrow s$.GetQueriedEntity()
2: Send(gqq, s_q)
3: Wait for other agents
4: $s_u \leftarrow$ Find(gqq, s.GetUpdatedEntity())
5: Send(gdq, s_u)
6: Wait for other agents and upper system synchronization
7: s.Update(Fetch(gdq, s_q))

3) Synchronization Skipping: Following the characteristics of distributed graph processing, it happens that some iterations can be skipped, so that the synchronization overheads of these iterations can be saved. The observation is that, there is no need to trigger the global synchronization, if there is no de facto "conflicts" among distributed nodes to be synchronized, i.e., updated data of a node is not needed by all other nodes. We therefore design a mechanism call "synchronization skipping" based on synchronization caching, to detect if the current iteration synchronization can be skipped.

As shown in Figure 7, at the end of cache updating, agent checks if each updated vertex and its outer edges are in the same node. If it is true for an agent, it means the agent can continue with the next iteration using its local data.

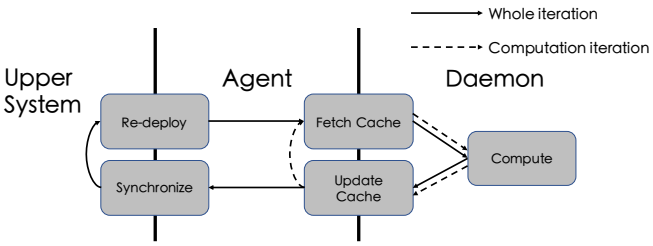


Fig. 7: Synchronization Skipping

If it is true for all agents, it means that there is no need for any inter-node data transferring. Thus, the upper system process can be skipped and next computation iteration can be directly started. This way, multiple computation iterations can be equivalent to a logically combined iteration, and therefore unnecessary synchronization for intermediate iterations can be totally skipped.

C. Beyond-Iteration Optimization: Workload Balancing

1) Motivation: As a "software glue", the middleware connects different accelerators and different upper systems. For instance, upper systems may adopt various graph partitioning strategies for assigning subgraphs to distributed nodes, which may cause storage imbalance. On the other hand, different accelerators may have different computation powers, which may cause computation imbalance. Therefore, it is important for the middleware to have a mechanism to detect and react to workload balancing, so that the performance of parallelism can be maximized.

2) Analysis: To this end, we introduce a simply yet effective workload estimation model for the middleware to predict the performance of data processing of a local node.

Suppose there are in total D data entities which are partitioned into m distributed nodes, satisfying $\sum_{j=1}^{m} d_j = D$. According to pipeline shuffle mechanism in Section III-A, the total processing time consists of three parts. For ease of discussion, for distribute node j, we set the total time cost T^j_{total} taken by Thread.Download, Thread.Compute, and Thread.Upload as T^j_n, T^j_c, and T^j_u, respectively, satisfying $T^j_{total} = T^j_n + T^j_c + T^j_u$.

Noticed that both T^j_n and T^j_u are proportional to d_j. T^j_c's computation time and data copying time are also proportional to d_i, and calling time T^j_{call} is proportional to the number of blocks s. We can have:

$$T^j_{total} = T^j_n + T^j_c + T^j_u = c_j \cdot d_j + s \cdot T^j_{call}$$

Here, c_j is the coefficient associated with node j to represent the relation between data size and process time. Since there is no relationship between s and d_j, there is no need to consider s in this situation. Then, given a set of m distributed nodes, the objective of workload balancing can be represented by:

$$\min(\max_{j \le m}(c_j \cdot d_j) \tag{6}$$

Here, we call $\frac{1}{c_j}$ as *computation capacity factor*, since c_j shows "time cost per unit amount of data", and $\frac{1}{c_j}$ shows "data processed per unit time".

3) Mechanism: With the objective function (Equation 6), we come up with two metrics for the middleware to detect and react to the workload imbalance. The first one offers benchmarks for upper systems to adjust partitioning strategies, given a specific configuration of accelerators for distributed nodes. The second one offers benchmarks for upper systems to supervise the assignment of accelerators to distributed nodes, under a specific graph partitioning strategy. In other words, our estimation model can be applied for 2 cases. The first case is on the tuning of $\{d_j\}$ with fixed $\{c_j\}$. The second case is on the tuning of $\{c_j\}$ with fixed $\{d_j\}$.

Case 1: tuning $\{d_j\}$ *under fixed* $\{c_j\}$. Lemma 2 makes the theoretical basis for getting optimal values of d_j.

Lemma 2. Given D data entities which are partitioned to m distributed nodes, where each node holds a data fragment d_j, satisfying $\sum_{j=1...m} d_j = D$, the balancing target is to minimize function $G(.)$, which represents the maximum time cost of a distributed node.

$$G(d_1, ..., d_m) = \max_{j=1...m} (c_j d_j)$$

It holds that function $G(.)$ achieves its minimum value, iff every element d_j of its m-dimensional input variable $\{d_j\}_{j \leq m}$ satisfies:

$$d_j = \frac{\frac{1}{c_j}}{\sum_{j=1}^{m} \frac{1}{c_j}} D$$

Proof. First, if every d_j meets the condition, we have:

$$G(d_1, ..., d_m) = \max_{j=1}^{m} \{c_j \cdot \frac{\frac{1}{c_j}}{\sum_{j=1}^{m} \frac{1}{c_j}} D\} = \frac{D}{\sum_{j=1}^{m} \frac{1}{c_j}}$$

Second, we prove that for any possible d_j, we have $G \geq \frac{D}{\sum_{j=1}^{m} \frac{1}{c_j}}$. We prove this assertion by contradiction. We first assume it holds that $G = max_{j=1}^{m}(c_j d_j) < \frac{D}{\sum_{j=1}^{m} \frac{1}{c_j}}$. Then for every d_j, we have:

$$d_j < \frac{\frac{1}{c_j}}{\sum_{j=1}^{m} \frac{1}{c_j}} D \Rightarrow D = \sum_{j=1}^{m} d_j < \frac{\sum_{j=1}^{m} \frac{1}{c_j}}{\sum_{j=1}^{m} \frac{1}{c_j}} D = D$$

Here, contradiction occurs. Thus, function F reaches the minimum value $\frac{D}{\sum_{j=1}^{m} \frac{1}{c_j}}$, if and only if for all d_j, $d_j = \frac{\frac{1}{c_j}}{\sum_{j=1}^{m} \frac{1}{c_j}} D$. The lemma is hence proved. □

Lemma 2 shows that $\frac{\frac{1}{c_j}}{\sum_{j=1}^{m} \frac{1}{c_j}}$ can serve as *balancing factor*s for selecting appropriate partitioning strategies. For example, given a set of partitioning strategies, the one that achieves minimum $F(.)$ is to be selected.

Case 2: tuning $\{c_j\}$ *under fixed* $\{d_j\}$. It is possible for upper systems to elastically select demanding number of accelerators (e.g., from a GPU cloud), given that the graph partitioning results are fixed (graph partitioning is more I/O and computational intensive than other processing phases). If so, the middleware can adjust the *computation capacity factor* $1/c_j^{'}$ for balancing the workload, according to Lemma 3.

Lemma 3. Given D data entities which are partitioned to m distributed nodes, where each node holds a data fragment d_j, satisfying $\sum_{j=1}^{m} d_j = D$, and given the maximum available computation capacity factor f ($f \geq \max_{j=1...m} \frac{1}{c_j}$), our target is to minimize function $G'(.)$, which indicates the maximum time cost of a distributed node.

$$G^{'}(\frac{1}{c_1}, ..., \frac{1}{c_m}) = \max_{j=1}^{m}(c_j d_j)$$

Function $G'(.)$ achieves its minimum value, if every element $\frac{1}{c_j}$ of its m-dimensional input variable $\{\frac{1}{c_j}\}_{j \leq m}$ satisfies:

$$\frac{1}{c_j} = \frac{f \cdot d_j}{d_*}, \qquad \textit{where } d_* = max_{j \leq m}(d_j)$$

Proof. Let $\frac{1}{c_*}$ be $max_{j \leq m} \frac{1}{c_j}$. Since $\frac{1}{c_*} \leq f$, we have:

$$\frac{d_*}{f} \leq c_* d_* \leq F^{'} = \max_{j=1}^{m}(c_j d_j)$$

To make $G^{'} = \frac{d_*}{f}$, all other $c_j d_j$ must be not greater than $\frac{d_*}{f}$. With the condition to make $\frac{1}{c_j}$ be as small as possible, we have:

$$\frac{1}{c_j} = min\{\frac{1}{c_j}, \text{where } c_j d_j \leq \frac{d_*}{f}\} = \frac{f d_j}{d_*}$$

Thus, the lemma is proved. □

According to Lemma 3, the middleware can dynamically allocate idle accelerators to generate more daemons for the node demanding more computation powers, as long as conditions of computation capacity factor of every partitions are met.

IV. System Implementation

We show details on key implementation of the middleware. We introduce the programming interfaces offered by the middleware in Section IV-A. We study System IPC-base communication between daemons and agents in Section IV-B. We investigate environment accessing in Section IV-C.

A. APIs

It is important for the middleware to create a series of easy-to-use interfaces to make accelerators plugged to upper systems easy and coder-friendly. In our implementation, we design an iteration-based graph algorithm template and an interface set to connect with upper systems.

1) Algorithm Template: The APIs of algorithm template follow a unified iterative model and support C++-based code integration, including OpenMP, MPI, CUDA and so on. There are 3 steps with computation of an iteration for general multiworker systems, *Message Passing*, *Combining* and *Aggregating*, in which external computation resources can be utilized for computation optimization. Our APIs contain 3 interfaces, in correspondence to the above 3 steps: MSGGen(), MSGMerge() and MSGApply(). With the help of the daemon-agent framework, runtime details, such as data transferring between

the three APIs, runtime order, interactions with upper layers and extra resource management details are hidden to algorithm engineers , thus they only focus on the implementation of three APIs for specific graph algorithms.

More, with the separate maintenance of APIs of MSGGen(), MSGMerge() and MSGApply(), upper system developers can arrange the API calls in different orders, so that the middleware is adaptable to various graph computation models, such BSP, GAS, and asynchronous model, shown in Section IV-C2.

2) Operation Interfaces: To make upper system calls easier to be adapted to agents, agent accessing is organized into a set of interface functions. The design of agent interfaces include three functions for daemon-agent communication, i.e., connect(), disconnect(), and shutdown(), two functions for data transferring, i.e., transfer() and, update(), and function requestX() for computation lifecycle controlling. Here, X can be anyone of the three APIs, MSGGen(), MSGMerge() or MSGApply(). Upper system developers only need to access corresponding functions by inputting proper parameters to get the full control of daemon runtime. Also, it takes merely a few lines of code to connect to upper systems.

In summary, with such interface functions of agents, computation daemons can be integrated and cooperated for the global computation invoked by upper systems.

B. System V IPC-based Deployment

Following the daemon-agent framework, data appeared in an agent given by upper systems can not be directly accessed by daemons, since they belong to different processes with no common memory space, as shown in Section IV-C3. Also, traditional inter-process communication approaches trigger extra data transfers, and significantly degrade the overall system performance.

To tackle this, we resort to kernel functions aided by UNIX System V. There is thus no extra memory overhead taken for daemons or agents to handle the data transferring. Instead, a daemon has a unique System V key, that represents a specific System V share memory space in a local distributed node, while an agent has multiple keys to communicate with all daemons attached to the agent. Agents and daemons can communicate with corresponding System V message queues, facilitating the sequence control of the system. Also, since data from upper systems or accelerators is kept in the System V share memory, any data updates between agent and daemons can be immediately perceived, without extra sensing efforts or intermediate transfers, yielding the minimum data transferring between the two ends. Furthermore, pipeline shuffle as shown in Section III-A), can be implemented by changing the order of internal pointers of System V space. Such changes can also be observed by agents for subsequent memory operations, without any extra communication with daemons.

C. Environment Accessing

1) GraphX (JVM) : JVM is a uniform environment separated from local environment of distributed nodes to execute Java programs. However, it makes things complicated when GraphX needs external tools or libraries for computation and other usages. JNI provided in JVM suffers from extra costs on invoking native target functions due to JNI callbacks, which can be eased by the native memory [21]. To solve the problem, we design 2 components to efficiently break the barrier between the JVM runtime and local environment.

JNI Transmitter. We design a set of JNI-based function and their related GraphX native method interfaces for natural hybrid system implementation. However, naively invoking JVM methods at runtime incurs significant transmission lags. Hence, we utilize a series of techniques such as POSIX-based shared memory data exchanging, batch data transferring, in JNI transmitter in order to reduce JNI calls.

Data Packager. Data packager solves the inconsistency of data structures of the two ends of JVM and local environment. This component uses techniques such as bit data organization, JVM data structure space reserving, to transform data between the two formats, with high efficiency, and without extra space usage and redundant data copying. Preliminary testing shows that about 3 to 10 times of improvement can be achieved, compared with native target function invoking.

2) PowerGraph : PowerGraph [3] follows another computation model, called Gather-Apply-Scatter (GAS *in short*), which is widely used by many distributed graph processing frameworks. Although BSP and GAS are of different graph computation models, they are common in basic iterative characteristics [22], which can be view as different orders of iterative operations, etc. It paves the road for theoretical and technical foundation of a general middleware design in supporting different computation models.

For example, when connecting to Powergraph, MSGGen(), MSGMerge() and MSGApply() is used to represent scatter, gather, and apply steps in GAS model. The execution order in PowerGraph thus follows Merge() $\rightarrow$ Apply() $\rightarrow$ Gen(), which differs from BSP model, i.e., Gen() $\rightarrow$ Merge() $\rightarrow$ Apply(). In an iteration, PowerGraph calls agent interfaces, in the order of requestMerge(), requestApply() and requestGen(), so that GAS model can be supported by the middleware without any extra code modifications. It shows the generality of our middleware in adapting to different computation models.

3) Runtime Isolation: As shown in Section II-A2, agent could be intuitively designed as a parent process of daemons for computation resource calls. If so, however, the daemon and computation device environment would be re-initialized multiple times during the iterative graph processing. Because the launching and ceasing of an agent must be triggered multiple times by the upper system, and so do their associated daemons. The frequent re-initialization incurs considerable system overheads, since the initialization process of daemons (with internal function calls) and associated computational devices is time-consuming.

To overcome the dilemma, our daemon-agent framework detaches the initialization process from direct function calls. Daemons and agents work as independent processes, and they communicate with each other by message exchange. This way,

a daemon never triggers re-initialization during the iterative graph processing.

V. Evaluation

A. Setup

We use both real and synthetic datasets for our experiments. For real data, we choose several representative public datasets commonly used for graph system testing to cover graphs of both cyber and physical worlds, including networks, social networks, road networks, etc, shown in Table I.

TABLE I: Datasets

Dataset	Vertex	Edge	Type
Synthetic dataset	4M	80M	Power-law
Orkut [23]	3.07M	117.18M	Social
Wiki-topcats [24]	1.79M	28.51M	Network
LiveJournal [25]	4.84M	68.99M	Social
WRN [26]	23.9M	28.9M	Road
Twitter [27], [28]	41.65M	1.468B	Social

We conduct experiments on a set of representative graph algorithms, such as Bellman-Ford (SSSP-BF), Connected Components (CC), PageRank (PR), and Label Propagation algorithm (LP), by varying datasets of different distributions and scales. For SSSP-BF, we use 4 vertices as source vertices and calculate their SSSPs simultaneously to make it more compute-intensive. For LP, we limit the iterations to 15 times to avoid unlimited computation on specific datasets. To test the performance on scaling up, experiments are run in a cluster, which has 6 physical computing nodes connected with 100Gbps Infiniband. Each of computing nodes has CPU Xeon E5-2698 v4 (2.20GHz, 20 cores), 2 NVIDIA V100 GPUs, 384GB memory, and a 1TB hard disk. For the middleware accelerator abstraction, we treat CPU in one node as an accelerator which has a 20-thread multithread processing model, and we treat each GPU as an accelerator which has 1024-thread multithread processing model.We build our system with Ubuntu 16.04.5 LTS and deploy a Nvidia-docker framework for simulating the distributed environment with heterogeneous processors. We construct a cross compilation solution by using maven, sbt and cmake to manage and config the global project dependencies. For Spark runtime, we use Java 8 and Scala 2.11. For local C++ and CUDA programming, g++7 and CUDA 10.0 are used. The source code is available in GX-Plug repository. Also, code for GraphX integration can be found at GraphXwithGPU repository, and code for PowerGraph integration can be found PowerGraph-GPU repository.

B. Results

1) Results on real graphs: Figure 8 compares the performance of GraphX and PowerGraph with non-acclerator(no prefix), CPU-integrated (prefix CPU+) and GPU-integrated (prefix GPU+) The y-axis is in log scale. In relatively computation-dense applications such as LP and SSSP-BF, acceleration can be observed in total time.

On GraphX, compared with GraphX, GPU+GraphX achieves up to 7x acceleration in SSSP-BF, and up to 20x acceleration in LP algorithm. CPU+GraphX also achieves up to 4x acceleration in SSSP-BF algorithm and up to 5x acceleration in LP algorithm. On PowerGraph, GPU+PowerGraph achieves up to 25x acceleration in SSSP-BF algorithm and up to 15x acceleration in LP algorithm. CPU+PowerGraph also achieves up to 5x acceleration in SSSP-BF algorithm and up to 10x acceleration in LP algorithm. These results verify the effectiveness of the middleware.

To evaluate the performance on scalability, we vary the number of nodes and GPUs invoked for fulfilling the computation task, as shown in Figure 9. It can be observed that sublinear speedup in computation time is achieved in GPU+. In particular, for WRN dataset (Figure 9 (b)), the time cost is about 1700s when 2 GPUs are allocated. The cost can be reduced to less than 1000s when 4 GPUs are used.

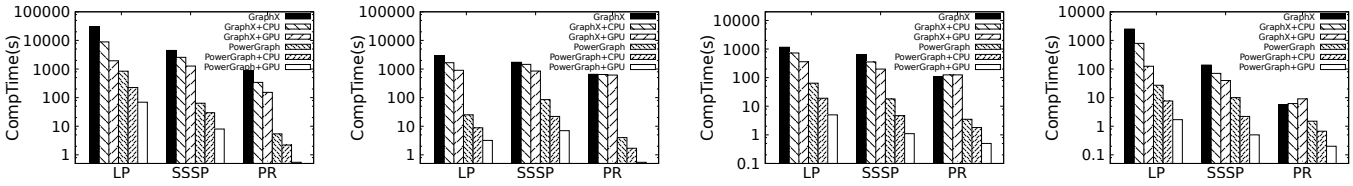
2) Effect of Pipeline Shuffle: Figure 10 shows the experiment results of the performance of pipeline shuffle mechanism on dataset Orkut. 3 competitors, "Pipeline*", "Pipeline" and "Without pipeline", in corresponding to the results with optimal blocksize, fixed blocksize, and the one without pipeline parallelism. Experiment shows that "Pipeline*" can achieve 30%-50% acceleration rate compared with "withoutPipeline". Also, "Pipeline*" can improve pipeline performance as 20%-30%, compared with "Pipeline".

3) Effect of Synchronization Caching & Skipping: Figure 11 (a) shows the performance of the synchronization caching mechanism. We use both synthetic and real graph datasets as input, and use SSSP-BF algorithm for testing the workload. The experiment result shows that we can get 2-3x acceleration in GraphX integrations. For the results on PowerGraph, it is much more efficient than GraphX. We can get up to 150% acceleration in both synthetic and real datasets.

Figure 11 (b) shows the performance of synchronization skipping mechanism. We use SSSP-BF algorithm for testing the workload, and count the number of iterations skipped on both synthetic and real datasets. We also compare the result with the number of iterations when synchronization skipping mechanism is disabled. For real datasets, the synchronization skipping mechanism achieves 60%-90% decrease of the number of iterations. However, the effect on synthetic dataset is insignificant, where the data are more uniform, due the random generation of nodes and edges. For real datasets, there tends to be more clusters of dense partitions, leading to better partitioning results that triggers synchronization skipping.

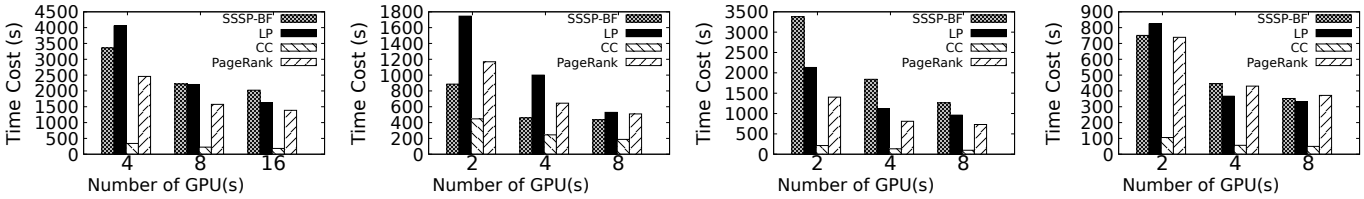
4) Effect of Workload Balancing: Figure 12 shows the results of workload balancing. We compare the difference of system performance with and without workload balancing. Also, we plot the best performance can be achieved in accordance to our estimation model as discussed in Section III-C.

Figure 12 (a) shows the scenario in which the hardware configuration of distributed nodes are fixed, and the partitioning strategy can be tuned (Case 1, Section III-C). We construct two distributed nodes for the experiment. One node contains 1 GPU, and the other contains 3 GPUs. We evenly partition the graph dataset to all nodes, which is the default setting of distributed graph systems, and is denoted as "Not Balanced".



(a) algos @ Twitter2010 (b) algos @ Orkut (c) algos @ LiveJournal (d) algos @ Wiki-topcats

Fig. 8: GPU + GraphX/PowerGraph Performance Benchmark on Real Datasets



(a) syn10M (b) WRN (c) LiveJournal (d) Wiki-topcats

Fig. 9: Results on Scalability (GraphX + GPU)

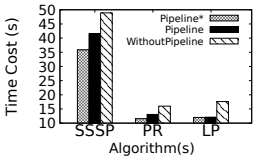
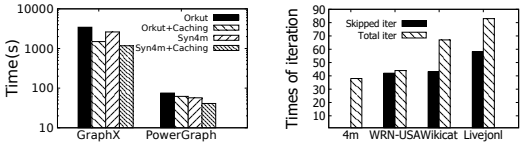
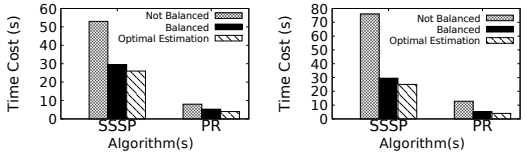


Fig. 10: Performance of Pipeline Shuffle



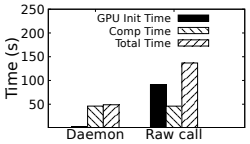
(a) Caching on Systems (b) # of Skipped Iterations

Fig. 11: Sync Cacheing & Skipping Performance

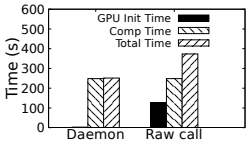


(a) Fixed Comp Resource (b) Fixed Data Partitioning

Fig. 12: Workload Balancing Performance

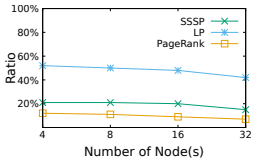


(a) SSSP-BF, syn1m

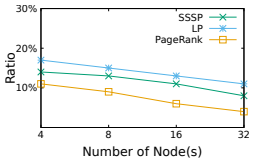


(b) SSSP-BF, WRN

Fig. 13: Runtime Isolation Performance

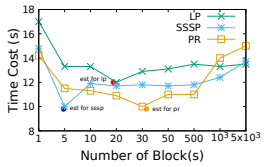


(a) PowerGraph, Orkut



(b) GraphX, Orkut

Fig. 14: Middleware Cost Ratio



(a) Estimation vs Real

Fig. 15: Estimating s_{opt}

We compare it to the one with our balancing strategy as discussed in Section III-C, which is denoted as "Balanced". It shows that the workload balancing can significantly improve the system performance. Also, the balanced result is very close to the theoretically optimal result.

Figure 12 (b) shows the scenario in which the partitioned results are fixed, and the hardware configuration can be tuned (Case 2, Section III-C). We construct 2 distributed nodes with the same hardware configuration. We vary the data load of distributed nodes to observe the effect of hardware configuration tuning. Without balancing techniques, both distributed nodes are with 1 GPU, denoted as "Not Balanced". With balancing techniques, we can estimate the number of GPUs needed in accordance to the data load and dynamically allocate appropriate number GPUs, denoted as "Balanced". It shows that the workload balancing can significantly improve the system performance. Also, the balanced result is very close to the theoretically optimal result, demonstrating the merits of workload balancing strategies.

5) Effect of Runtime Isolation: We hereby examine the performance of computation daemon on the runtime isolation mentioned in Section IV-C3. We design a comparative test to compare the influence of GPU initialization between daemon-agent based solution and direct GPU call solution. We design two experiments running the same algorithm in different datasets to create different numbers of iterations. A larger number of iterations corresponds to a higher number of times of CPU-GPU runtime environment switching.

Results in Figures 13 (a) (32 iterations) and 13 (b) (43 iterations) show that our solution significantly reduces unnecessary initialization cost. The benefits would be amplified when # of iterations is higher.

6) Middleware Scalability: We examine the scalability of our middleware, by varying the number of distributed nodes, in Figure 14. It plots the ratio of time cost taken by the middleware to the cost of the entire system, for different graph tasks on different distributed systems, e.g., PowerGraph and GraphX. It can be observed that, for all graph tasks, the time ratio of the middware decreases w.r.t. the increase of

number of distributed nodes. The downhill trend reflects good scalability of the middleware in a larger scaled distributed computing environment, where the cost can be dominated by the gradually enlarged synchronization overhead of distributed system side. Also, the time ratios of middleware are mostly between 10% and 20%, especially for algorithms with high operational intensities. Particularly, PageRank takes only about 10% of total cost in a distributed system with 32 nodes. LP is different, since it is a fully iterative algorithm, corresponding to a low operational intensity.

In summary, the low cost ratio and the downhill trend demonstrate good scalability of our middleware.

7) Block Size Selection: To examine the performance improvement brought by block size selection, we design a experiment shown as Figure 15 to measure the pipeline performance in different with different amount of blocks s. we also calculate $s_{opt} = \frac{d}{b_{opt}}$ estimation and use it to compare with the real performance[2].

For both LP and PageRank algorithms, we use the first iteration as the testing data. For SSSP algorithm, we use 6-th iteration as the testing data, since the computation workload is the maximum during the whole execution. We can find that when s increases, iteration time cost first decreases, and then increases. Thus, for $b = \frac{d}{s}$, when b increases, iteration time cost also tends to first decrease, and then increase.

We also give our estimated s following the analysis here for the 3 different algorithms. It shows that when real b and s are close to our estimation, the pipeline performance is also close to our estimation, which shows the accuracy of our estimation. Also, optimal performance can be reached when real b and s are close to our estimation value, which shows the correctness of our estimation.

VI. Related Works

Distributed CPU-based Systems. With the prosperity of distributed system, people investigate common operator sets inside diverse graph primitives for scaling out in the distributed environment. As a forerunner, Pregel [11] is proposed by Google on large-scale graph computing, following the BSP model. In BSP, graph computation are divided into iterations and intermediate results can be globally synchronized at barriers called super-steps. GraphLab [29] allows asynchronous computation and dynamic asynchronous scheduling, whose programming model also isolates the user-defined algorithm from data movement. To achieve better workload balancing on natural graphs, PowerGraph [3] uses a more flexible GAS abstraction for power-law graphs.

There are also many embedded graph processing systems built on existing distributed systems to gain the benefits of task scheduling and data management. GraphX [2] is one of the most successful representative built on top of Apache Spark [30]. HaLoop [31] is a similar distributed graph processing system, in particular, extended from Hadoop [32].

However, most works pay little attention to the computation intensiveness of large-scale graph processing. Efforts on scheduling balancing and data accessing also incur extra cost in computation, making the system even slower than single-node solutions. It is thus desired to have a scale-up solution for distributed graph systems.

Single-node Parallel Graph Algorithms. There also exist hardwired graph primitive implementations for the single-node environment. Merrill et al. propose linear parallelization of BFS algorithm on GPU [33]. Soman et al. studies graph algorithms based on two PRAM connected-component [34]. Several parallel Betweenness Centrality implementations are available on GPU based on the work of Brandes et al. [35]. Davidson et al. [36] propose a work-efficient Single-Source Shortest Path algorithm on GPU.

Low-level graph parallel solutions can have best performance only on specific computation tasks, but are not general for diversified graph applications. Also, the hardwired primitives are challenging to even skilled algorithm engineers, making such solutions hard for being deployed in real systems and applications.

High-level GPU Programming Model. There are also existing works on high-level graph operations for GPU. Zhong and He devise Medusa [37] on a high-level GPU-based system for parallel graph processing using a message-passing model, which is arguably the earliest work for GPGPU development for graphs. CuSha [38], targeting a GAS abstraction, avoids non-coalesced memory accessing and avoids irregular memory accessing. Gunrock [4] implements a novel data-centric abstraction centered on operations on a vertex or edge frontier rather than designing an abstraction for computation. Recently, there are a few works on GPU graph processing system built on distributed systems, among which Lux [17] is one of the representatives. Users can use GPUs in multiple physical nodes for efficient computation. However, without the support of mature distributed systems, Lux faces a series of challenges, such as robust distributed data management, scheduling balancing, effective fault recovery, and efficient data synchronization with physical layers, and thus falls short in addressing technical issues arise in large-scale graph data management and analytics.

VII. Conclusion

In this paper, we propose a middleware for the integration of heterogenous distributed graph systems and accelerators. Our middleware is versatile in the sense that it supports different programming models, computation models, and runtime environments. For reinforcing the middleware performance, we devise a series of techniques, such as pipeline shuffle, synchronization caching and skipping, partitioning, and parameter configuration for intra-, inter, and beyond iteration optimization. Extensive experiments show that our middleware achieves good performance in large-scale graph processing.

[2]Coefficients are tested as follows: for SSSP: (k_1, k_2, k_3, a) = (0.03, 0.51, 0.09, 84671); for PR: (k_1, k_2, k_3, a) = (0.02, 0.58, 0.1, 1970); for SSSP: (k_1, k_2, k_3, a) = (0.003, 0.59, 0.006, 498).

REFERENCES

[1] K. Bouchard, J. Aimone, M. Chun, T. Dean, M. Denker, M. Diesmann, D. Donofrio, L. Frank, N. Kasthuri, C. Koch, O. Ruebel, H. Simon, F. Sommer, and M. Prabhat, "High-performance computing in neuroscience for data-driven discovery, integration, and dissemination," *Neuron*, vol. 92, pp. 628–631, 11 2016.

[2] 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 *OSDI*, J. Flinn and H. Levy, Eds., 2014, pp. 599–613.

[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *OSDI*, 2012, pp. 17–30.

[4] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: a high-performance graph processing library on the GPU," in *PPoPP*, 2016, pp. 11:1–11:12.

[5] X. Chen, H. Tan, Y. Chen, B. He, W. Wong, and D. Chen, "Thundergp: Hls-based graph processing framework on fpgas," in *FPGA '21: The 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Virtual Event, USA, February 28 - March 2, 2021*, L. Shannon and M. Adler, Eds. ACM, 2021, pp. 69–80. [Online]. Available: https://doi.org/10.1145/3431920.3439290

[6] S. Sakr, A. Bonifati, H. Voigt, A. Iosup, K. Ammar, R. Angles, W. G. Aref, M. Arenas, M. Besta, P. A. Boncz, K. Daudjee, E. D. Valle, S. Dumbrava, O. Hartig, B. Haslhofer, T. Hegeman, J. Hidders, K. Hose, A. Iamnitchi, V. Kalavri, H. Kapp, W. Martens, M. T. Özsu, E. Peukert, S. Plantikow, M. Ragab, M. Ripeanu, S. Salihoglu, C. Schulz, P. Selmer, J. F. Sequeda, J. Shinavier, G. Szárnyas, R. Tommasini, A. Tumeo, A. Uta, A. L. Varbanescu, H. Wu, N. Yakovets, D. Yan, and E. Yoneki, "The future is big graphs: a community view on graph processing systems," *Commun. ACM*, vol. 64, no. 9, pp. 62–71, 2021.

[7] Amazon, "Amazon ec2 instance types," https://aws.amazon.com/ec2/instance-types/?nc1=h_ls, 2021.

[8] NVIDIA, "Gpu-accelerated apache spark," https://www.nvidia.com/en-us/deep-learning-ai/solutions/data-science/apache-spark-3/, 2020.

[9] Databrick, "Deep dive into gpu support in apache spark 3.x," https://databricks.com/session_na20/deep-dive-into-gpu-support-in-apache-spark-3-x, 2020.

[10] K. Ammar and M. T. Özsu, "Experimental analysis of distributed graph systems," *Proc. VLDB Endow.*, vol. 11, no. 10, pp. 1151–1164, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol11/p1151-ammar.pdf

[11] 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 *SIGMOD*, 2010, pp. 135–146.

[12] X. Chen, "Graphcage: Cache aware graph processing on gpus," *CoRR*, vol. abs/1904.02241, 2019. [Online]. Available: http://arxiv.org/abs/1904.02241

[13] P. Wang, J. Wang, C. Li, J. Wang, H. Zhu, and M. Guo, "Grus: Toward unified-memory-efficient high-performance graph processing on GPU," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 2, pp. 22:1–22:25, 2021. [Online]. Available: https://doi.org/10.1145/3444844

[14] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, 2020.

[15] NVIDIA, "Nvlink and nvswitch: The building blocks of advanced multi-gpu communication," https://www.nvidia.com/en-us/data-center/nvlink/, 2021.

[16] H. Fu, M. G. Venkata, S. Salman, N. Imam, and W. Yu, "Shmemgraph: Efficient and balanced graph processing using one-sided communication," in *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018*, E. El-Araby, D. K. Panda, S. Gesing, A. W. Apon, V. V. Kindratenko, M. Cafaro, and A. Cuzzocrea, Eds. IEEE Computer Society, 2018, pp. 513–522. [Online]. Available: https://doi.org/10.1109/CCGRID.2018.00078

[17] Z. Jia, Y. Kwon, G. M. Shipman, P. S. McCormick, M. Erez, and A. Aiken, "A distributed multi-gpu system for fast graph processing," *VLDB*, vol. 11, no. 3, pp. 297–310, 2017.

[18] R. Albert, H. Jeong, and A.-L. Barabási, "Error and attack tolerance of complex networks," *Nature*, vol. 406, no. 6794, pp. 378–382, 2000.

[19] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 17–30.

[20] D. Donato, L. Laura, S. Leonardi, and S. Millozzi, "Large scale properties of the webgraph," *The European Physical Journal B*, vol. 38, no. 2, pp. 239–243, 2004.

[21] N. A. Halli, H. Charles, and J. Méhaut, "Performance comparison between java and JNI for optimal implementation of computational micro-kernels," *CoRR*, vol. abs/1412.6765, 2014. [Online]. Available: http://arxiv.org/abs/1412.6765

[22] M. T. Özsu, "Graph processing: A panoramic view and some open problems," *VLDB*, 2019.

[23] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and Analysis of Online Social Networks," in *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07)*, San Diego, CA, October 2007.

[24] H. Yin, A. R. Benson, J. Leskovec, and D. F. Gleich, "Local higher-order graph clustering," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*. ACM, 2017, pp. 555–564. [Online]. Available: https://doi.org/10.1145/3097983.3098069

[25] L. Backstrom, D. P. Huttenlocher, J. M. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*, T. Eliassi-Rad, L. H. Ungar, M. Craven, and D. Gunopulos, Eds. ACM, 2006, pp. 44–54. [Online]. Available: https://doi.org/10.1145/1150402.1150412

[26] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: http://networkrepository.com

[27] P. Boldi and S. Vigna, "The webgraph framework I: compression techniques," in *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, Eds. ACM, 2004, pp. 595–602. [Online]. Available: https://doi.org/10.1145/988672.988752

[28] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, Eds. ACM, 2011, pp. 587–596. [Online]. Available: https://doi.org/10.1145/1963405.1963488

[29] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," *VLDB*, vol. 5, no. 8, pp. 716–727, 2012.

[30] "Spark," http://spark.apache.org, 2019.

[31] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "The haloop approach to large-scale iterative data analysis," *VLDB J.*, vol. 21, no. 2, pp. 169–190, 2012.

[32] "Hadoop," http://hadoop.apache.org, 2019.

[33] D. Merrill, M. Garland, and A. S. Grimshaw, "Scalable GPU graph traversal," in *PPoPP*, 2012, pp. 117–128.

[34] J. Greiner, "A comparison of parallel algorithms for connected components," in *SPAA*, 1994, pp. 16–25.

[35] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, 2001.

[36] A. A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel GPU methods for single-source shortest paths," in *IPPS*, 2014, pp. 349–359.

[37] J. Zhong and B. He, "Medusa: A parallel graph processing system on graphics processors," *SIGMOD Record*, vol. 43, no. 2, pp. 35–40, 2014.

[38] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: vertex-centric graph processing on gpus," in *HPDC*, 2014, pp. 239–252.