UniGPS: A Unified Programming Framework for Distributed Graph Processing

Zhaokang Wang*†, Junhong Li*†, Yifan Qi*†, Guanghui Zhu*†, Chunfeng Yuan*†, Yihua Huang*†
**State Key Laboratory for Novel Software Technology, Nanjing University*, Nanjing, P. R. China
†*Department of Computer Science and Technology, Nanjing University*, Nanjing, P. R. China
{wangzhaokang,mg1833036,mg20330046}@smail.nju.edu.cn, {zgh,cfyuan,yhuang}@nju.edu.cn

***Abstract*—The industry and academia have proposed many distributed graph processing systems. However, the existing systems are not friendly enough for casual users like data analysts and algorithm engineers. On one hand, the programing models and interfaces differ a lot in the existing systems, leading to high learning costs and program migration costs. On the other hand, these graph processing systems are tightly bound to the underlying distributed computing platforms, requiring uses to be familiar with the details of distributed computing. To improve the usability of distributed graph processing, we propose a unified distributed graph programming framework UniGPS. We propose a unified cross-platform graph programming model VCProg for UniGPS. VCProg hides details of distributed computing from users. It is compatible with the popular graph programming models Pregel, GAS, and Push-Pull. VCProg programs can be executed by compatible distributed graph processing systems without modification, reducing the learning overheads of users. UniGPS supports Python as the programming language. We propose an interprocess-communication-based execution environment isolation mechanism to enable Java/C++-based systems to call user-defined methods written in Python. The experimental results show that UniGPS enables users to process big graphs beyond the memory capacity of a single machine without sacrificing usability. UniGPS shows near-linear data scalability and machine scalability.**

***Index Terms*—programming framework, distributed graph processing, graph processing systems, Python**

I. Introduction

A. Background

The graph is a useful data structure that can model relationships between multiple entities in the real world, such as social networks and transaction graphs in e-commerce platforms. The industry and academia have proposed many single-machine graph processing systems [1] to facilitate graph analysis. They provide users with friendly programming models and application programming interfaces (API). However, the computing power and memory space of a single machine limit their performance. As graph scales in real-world applications grow rapidly, their performance on big graphs becomes more and more unsatisfactory.

In order to process big graphs efficiently, many distributed graph processing systems are proposed [1]. They simplify distributed graph processing by providing users with high-level graph programming models and APIs. However, they are still not easy to use for users not familiar with distributed computing (like data analysts and algorithm engineers). They have the following shortcomings in usability.

- The programming interfaces of the existing systems lack the cross-platform feature. Each system has its unique programming model and APIs. The programs written for a certain system can only run on that system. When a new system with higher performance appears, users need manually migrate the existing programs to the new system, introducing additional learning costs and programming costs.
- The programming interfaces of the mainstream systems (like Giraph [2] and GraphX [3]) are tightly bound to the underlying distributed computing platforms. Users have to learn how to use distributed computing platforms before using the distributed graph processing systems, creating extra barriers. For example, Giraph requires users to be familiar with Hadoop's APIs such as Writable and FileInputFormat to write correct Giraph programs.
- Few mainstream systems support Python as the programming language. The mainstream systems (Giraph, GraphX, and Gemini) only support compiled languages (Java, Scala, and C++), but the data analysts or algorithm engineers prefer using Python [4] [5]. Python, along with the interactive development environment Jupyter Notebook, is more suitable for data exploration than the compiled languages.

In order to make graph programming easier, a distributed graph programming framework should satisfy three usability criteria: 1) provide a cross-platform unified programming interface, 2) make distributed computing details transparent to users, and 3) support Python as the programming language.

Unfortunately, as shown in Table I, the mainstream distributed graph processing systems hardly meet the three criteria simultaneously. Although KDT [9] satisfies the three criteria, the expression power of its linear algebraic programming model is limited, only supporting user-defined functions in several semiring methods like SpGEMM. TinkerPop [10] is a cross-platform graph programming framework, but it only supports Pregel as the programming model for *graph processing*. It cannot integrate with systems that adopt other programming models (like GraphX and Gemini). Furthermore, it only supports Java for graph processing.

B. Contributions

To enhance the usability of distributed graph processing, we propose a cross-platform unified distributed graph program-

TABLE I
COMPARISON OF DISTRIBUTED GRAPH PROCESSING SYSTEMS/FRAMEWORKS

System/Framework	Programming Model	Underlying Platform	Programming Language	Distributed Transparency	Interactive Execution	Development Environment
Giraph [2]	Pregel	Hadoop	Java	×	×	IDE
GraphX [3]	GAS	Spark	Scala	×	✓	IDE + Notebook
Gemini [6]	Push-Pull	MPI	C++	×	×	IDE
PowerGraph [7]	GAS	MPI	C++	×	×	IDE
PowerLyra [8]	GAS	MPI	C++	×	×	IDE
KDT [9]	Linear Algebra	MPI	Python	✓	✓	IDE + Notebook
TinkerPop [10]	Pregel	Multiple	Java	✓	×	IDE
UniGPS	VCProg	Multiple	Python	✓	✓	IDE + Notebook

ming model *VCProg*. We summarize the common features of the typical graph programming models Pregel, GAS, and Push-Pull and further propose the VCProg programming model to unify them. VCProg is vertex-centric. It regards graph processing as an iterative update of vertex properties. Each iteration consists of three phases: merging messages, updating vertex properties, and sending messages. VCProg is compatible with Pregel, GAS, and Push-Pull models. Programs written with VCProg can be executed by the compatible distributed graph processing systems (like Giraph, GraphX, and Gemini) without modification, achieving the goal of "Write Once, Run Anywhere."

Based on the unified programming model VCProg, we further design a unified distributed graph programming framework *UniGPS* that satisfies the three usability criteria.

1) UniGPS provides users with unified programming interfaces. Programs written with UniGPS can be executed by the mainstream distributed graph processing systems Giraph, GraphX, and Gemini without modification.
2) The programming interfaces are platform-independent, hiding details of the underlying distributed computing from users.
3) UniGPS adopts Python as the programming language. To enable the Java/C++-based distributed graph processing systems to call user-defined functions written in Python, we propose an execution isolation mechanism based on interprocess communication. We propose the zero-copy optimization technique based on memory-mapped buffers to reduce data copying overheads during interprocess communication.

We evaluate the usability and performance of UniGPS in a cluster with nine nodes. UniGPS enables users to conduct distributed graph processing in a user-friendly interactive Python development environment like Jupyter Notebook. With the help of distributed computing, the graph scale that UniGPS can handle is an order of magnitude higher than that of the serial graph processing library NetworkX [11]. For the same graph, the processing time of UniGPS is much shorter than NetworkX. UniGPS achieves near-linear data and machine scalability for typical graph algorithms.

II. RELATED WORK

A. Distributed Graph Processing Systems

A series of distributed graph programming models and processing systems have been proposed to reduce the difficulty of distributed graph processing. Reference [1] presents a comprehensive overview of distributed graph programming frameworks. According to the granularity of parallel computing, the existing graph programming models can be divided into three categories: vertex-centric [12], block-centric [13], and subgraph-centric [14] [15]. The vertex-centric models are the most thoroughly studied, typical models including Pregel [16], GAS [7], and Push-Pull [6]. However, even the same programming model has different APIs in different systems, lacking the cross-platform feature.

TinkerPop [10] is a graph programming framework that integrates multiple graph databases and processing systems with unified programming interfaces. TinkerPop proposes Gremlin [17] as its platform-independent query language. However, TinkerPop's graph processing framework GraphComputer only supports Java as the programming language and adopts Pregel as the graph programming model, not compatible with GAS, Push-Pull, and other programming models. It can only integrate the Pregel-based graph processing systems.

B. Graph Processing Libraries for Python

Python is popular among data analysts [4] [5] due to its high usability and development efficiency. Many Python graph processing libraries have been developed for the single-machine environment [18]. NetworkX [11] and graph-tool [19] are two popular libraries [20]. However, their performance is limited by the computing power and memory capacity of a single machine. They can hardly process large-scale graphs.

A possible way to process big graphs in Python is to use general-purpose distributed computing systems like Dask and PySpark. Due to the lack of specialized encapsulation and optimization for graph processing, users have to manually manage graph data and message exchange between vertices, increasing the programming difficulty.

III. UNIFIED GRAPH PROGRAMMING MODEL

Among the existing distributed graph programming models, the vertex-centric models (like Pregel [16] and GAS [7])

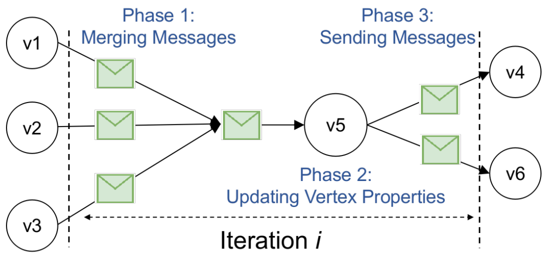

Fig. 1. Three phases of an iteration in vertex-centric programming models.

```
class VCProg:
  // Generate initial property for each vertex
  def initVertexAttr(id, out_degree, prop):
    return init_prop;
  // Get the empty message
  def emptyMessage():
    return empty_msg;
  // Merge two messages
  def mergeMessage(m1, m2):                                 Phase 1
    return msg;
  // Update the property of a vertex
  def vertexCompute(prop, msg, iter):                       Phase 2
    return new_prop, is_active;
  // Send a message along an outgoing edge of src
  def emitMessage(src, dst, src_prop, edge_prop):           Phase 3
    return is_emit, msg;
```

Fig. 2. Application programming interface of VCProg in Python.

are the most thoroughly studied [1]. However, different programming models have different APIs, increasing the learning burden and bringing extra programming migration costs. In order to overcome the drawback, we summarize the common features of three typical vertex-centric graph programming models Pregel [16], GAS [7], and Push-Pull [6]. We design a vertex-centric unified graph programming model *VCProg* based on the common features.

A. Common Features of Vertex-Centric Programming Models

Pregel, GAS, and Push-Pull have two salient common features: iterative calculation and three-phase update. The programming models express graph processing as an iterative update process of vertex properties. Vertices exchange data by sending messages between iterations.

The whole process consists of several rounds of iterations. In each iteration, every vertex updates its property based on the properties of itself, its neighbors, and the received messages. The iterations continue until convergence. There are two popular convergence conditions: all vertices are inactive, or the iteration reaches the maximum number of rounds.

In each iteration, the process of every vertex consists of three phases: merging messages, updating vertex properties, and sending messages, as shown in Fig. 1. Firstly, each vertex receives messages from its incoming neighbors and merges the messages into a single message. Secondly, each vertex updates its property based on the merged message and its current property. Finally, each vertex sends messages to its outgoing neighbors based on its updated property and the properties of its adjacent outgoing edges.

Inspired by the features, we propose a vertex-centric unified graph programming model *VCProg* that is compatible with Pregel, GAS, and Push-Pull at the same time.

B. Data Model of VCProg

VCProg adopts the property graph as its data model. Each vertex (edge) has an attached property that is a record containing several fields. All vertex (edge) properties have the same schema. Messages exchanged between vertices are also records. All messages have the same schema.

Before iterations begin, each vertex (edge) initializes its property based on the input data. During iterations, the vertex properties are updated while the edge properties remain unchanged. After iterations, the vertex properties store the processing results. The vertex properties are output to files in a tabular form.

C. Application Programming Interface of VCProg

VCProg provides its Python API in the form of an abstract base class `VCProg` as shown in Fig. 2. All methods of the base class are abstract. To write distributed graph processing programs, users need to inherit the base class and implement the abstract methods according to algorithmic logic. Users define the behavior of the three phases in each iteration with the `mergeMessage`, `vertexCompute`, and `emitMessage` method, respectively.

The `mergeMessage` method combines two message records `m1` and `m2` into a single message record `msg`. The message order should be interchangeable: `mergeMessage(m1, m2) = mergeMessage(m2, m1)`.

The `vertexCompute` method generates the updated vertex property for a vertex in each iteration. It receives the vertex property `prop` from the previous iteration, the merged message `msg`, and the current iteration number `iter` as the parameters. The method returns the updated vertex property `new_attr` and a flag `is_active` to indicate whether the vertex will be active in the next iteration.

The `emitMessage` method determines whether to send a message `is_emit` and the content of the message `msg` for the edge (`src`, `dst`), based on the source vertex's property `src_prop` and the edge's property `edge_prop`.

VCProg further uses the `initVertexAttr` method to initialize the property of each vertex before iterations, based on the vertex ID `id`, the property in the input `prop` and the outgoing degree `out_degree`. The `emptyMessage` method returns a global read-only empty message record `empty_msg`. The empty message is a special message that is idempotent for merging. For any message `m`, it should satisfy `mergeMessage(m, empty_msg) = m`.

Fig. 3 shows a demo program of using VCProg to implement the Bellman-Ford single-source shortest path calculation (UniSSSP). The API of VCProg is easy to use and independent from distributed graph processing systems, which ensures the transparency of distributed processing to users.

```python
import UniGPS, UniGPS.VCProg # Import the UniGPS library
# Initialize the UniGPS handle
unigps = UniGPS.createByHdfsConfFile(...)
# Load the input graph from a HDFS directory
in_graph = unigps.UniGraph.createByHdfsDir(path_to_input)
# Use the VCProg API of UniGPS
class UniSSSP(VCProg): # Implement SSSP with the VCProg base class
  self.ROOT = 0 # Source vertex
  def initVertexAttr(self, id, out_degree, prop):
    if vid == self.ROOT: # Construct a vertex property record
      self.vertexBuilder.setLong("vid", id).setLong("distance", 0)
    else:
      self.vertexBuilder.setLong("vid", vid).setLong("distance", sys.max)
    return self.vertexBuilder.build()
  def emptyMessage(self):
    self.msgBuilder.setLong("distance", sys.maxsize)
    return self.msgBuilder.build()
  def mergeMessage(self, m1, m2):
    aDis = m1.getLong("distance")
    bDis = m2.getLong("distance")
    self.msgBuilder.setLong("distance", min(aDis, bDis))
    return self.msgBuilder.build() # Construct a message record
  def vertexCompute(self, prop ,msg, iter):
    vDis = prop.getLong("distance")
    msgDist = msg.getLong("distance")
    isActive = False
    if vDis > msgDist:
      isActive = True
      prop.setLong("distance", msgDist)
    if iter == -1 and prop.getLong("vid") == self.ROOT:
      isActive = True
    return prop, isActive # Return updated vertex property record
  def emitMessage(self, src, dst, src_prop, edge_prop):
    srcDis = src_prop.getLong("distance")
    edgeWeight = edge_prop.getLong("weight")
    if srcDis == sys.maxsize:
      isEmit = False
      self.msgBuilder.setLong("distance", sys.maxsize)
    else:
      isEmit = True
      self.msgBuilder.setLong("distance", srcDis + edgeWeight)
    return isEmit, self.msgBuilder.build() # Construct a message record
# Execute the user program with the Giraph backend
out_graph = unigps.vcprog(in_graph, user_program=UniSSSP(),
                          engine="giraph",output_file=out_path)
# Use the native operator API of UniGPS
out_graph = unigps.sssp(in_graph, output_file=out_path, engine="giraph",
                        root=0)
out_graph.storeToDB(db_conf) # Save the output graph to a database
```

Fig. 3. Demo program of UniGPS to calculate the single-source shortest path (SSSP) in Python.

D. Execution Semantics of VCProg

Algorithm 1 shows the workflow of VCProg. VCProg receives a graph $G = (V, E)$ as the input, where V and E are the vertex and edge set of G, respectively. Each vertex v (edge e) in G has an attached property $v.value$ ($e.value$). The user needs to specify the maximum number of iterations *MAX_ITER* as the hyper-parameter. The user provides its program in the form of an instance object *VP* of the VCProg base class. *VP* implements all the abstract methods. *VP* is read-only and shared by all vertices.

VCProg iterates for at most *MAX_ITER* rounds (line 10). In each round, every vertex v will be either active or inactive. v is active if and only if v is set as active in the previous round or v receives any message. Only active vertices update their properties (line 10). The `vertexCompute` method determines whether the vertex remains active in the next round. If all vertices are inactive in a round (line 18), the iteration converges and terminates early. VCProg triggers a global barrier implicitly at the end of each round.

Algorithm 1 Workflow of the VCProg Programming Model

Input: $G = (V, E)$, *MAX_ITER*, *VP*
Output: G

1: $empty\_msg \leftarrow$ *VP.emptyMessage*(); ▷ Global empty message
2: **for all** $v \in V$ **do in parallel** ▷ Initialize vertex properties
3: $v.value \leftarrow$ *VP.initVertexAttr*(v.*ID*, len($v.out\_edges$), $v.value$);
4: **for** $iter \leftarrow 1$ to *MAX_ITER* **do**
5: $num\_active \leftarrow 0$; ▷ Number of active vertices
6: **for all** $v \in V$ that v is active or v receives any message **do in parallel**
7: $msg \leftarrow empty\_msg$;
8: **for** every message m received by v **do**
9: $msg \leftarrow$ *VP.mergeMsg*(msg, m);
10: $v.value$, $v.is\_active$ $\leftarrow$ *VP.vertexCompute*($v.value$, msg, $iter$);
11: **if** $v.is\_active$ = TRUE **then** ▷ Only for active vertices
12: $num\_active \leftarrow num\_active + 1$;
13: **for all** $e \in v.out\_edges$ **do** ▷ For outgoing adjacent edges
14: is_emit, msg $\leftarrow$ *VP.emitMessage*(v.*ID*, e.*target_ID*, $v.value$, $e.value$);
15: **if** is_emit = TRUE **then**
16: *// Implemented by the graph processing system*
17: SENDMESSAGE(e.*target_ID*, msg)
18: **if** $num\_active = 0$ **then**
19: **break;** ▷ The iteration converges

E. Compatibility with Other Programming Models

VCProg is compatible with the typical vertex-centric graph programming models Pregel [16], GAS [7] and Push-Pull [6]. The workflow of VCProg can be equivalently expressed by these models. Fig. 4 shows how to use Pregel, GAS, and Push-Pull to achieve the same execution semantics of VProg given a user-defined VCProg instance object *VP*. According to the conversion, the corresponding distributed graph processing system can execute a VCProg program *VP* to get the desired output. By this way, VCProg provides the capability of cross-platform execution.

IV. Unified Graph Programming Framework

A casual-user-friendly programming framework for distributed graph processing should hold three features: 1) provide Python APIs, 2) keep distributed processing transparent to users, and 3) support cross-platform program execution. Based on the unified graph programming model VCProg, we design and implement a unified graph programming framework *UniGPS*. UniGPS can be run in *interactive* Python development environments like Jupyter Notebook. UniGPS proposes an interprocess-communication-based mechanism to isolate the execution of user programs (written in Python) from the underlying distributed graph processing systems.

A. Application Programming Interface of UniGPS

UniGPS appears as a library in Python. Fig. 3 shows a demo program of using UniGPS to conduct the single-source shortest path analysis. Users need to import the library first and initialize a handle `unigps` of UniGPS with the configuration file. UniGPS supports loading/saving graphs from/to external data sources (like HDFS and graph databases).

With a loaded graph, UniGPS provides two kinds of APIs to conduct graph processing: 1) VCProg APIs that allows users developing customized processing programs, and 2)

```
Interface PregelVertexProgram // Express VCProg with the Pregel Vertex Program
  COMPUTE (v, messages):
      superstep ← GET_SUPERSTEP(); edges ← GET_OUTEDGES();
      if superstep = 0 then
          v.value ← VP.initVertexAttr(v.ID, len(edges), v.value);
      end if
      msg ← VP.emptyMessage();
      for m ∈ messages do
          msg ← VP.mergeMessage(msg, m);
      end for
      v.value, is_active ← VP.vertexCompute(v.value, msg, superstep);
      if is_active = FALSE or superstep ≥ MAX_ITER - 1 then
          VOTE_TO_HALT(v);
          return;
      end if
      for e ∈ edges do
          is_emit, msg ← VP.emitMessage(v.ID, e.target, v.value, e.value);
          if is_emit = TRUE then
              SEND_MESSAGE(e.target, msg);
          end if
      end for
```

(a) Pregel

```
Interface GASProgram (VP) // VP is an instance object of VCProg
// The global varialbe iter records the number of iterations
  GATHER (u, e, v):
    return e.msg;
  SUM (a,b):
    return VP.mergeMessage(a,b);
  APPLY (v, accum):
      v.value, v.is_active ← VP.vertexCompute(v.value, accum, iter);
      return v;
  SCATTER (v, e, u):
      e.msg ← VP.emptyMessage();
      if v.is_active = TRUE and iter < MAX_ITER then
          is_emit, msg ← VP.emitMessage(v.ID, u.ID, v.value, e.value);
          if is_emit = TRUE then
              e.msg ← msg;
      return e.msg;
```

(b) GAS

```
// The global varialbe iter records the number of iterations;
ProcessVertices:
  WORK (v):
      v.value, v.active ← VP.vertexCompute(v.value, v.msg, iter);
    return v.value;
ProcessEdges:
  DENSESIGNAL (v, inEdgeIterator):
      sum ← VP.emptyMessage();
      for e ∈ inEdgeIterator do
          emit, msg ← VP.emitMessage(e.src, v, e.src.value, e.value)
          if emit = TRUE then
              sum ← VP.mergeMessage(sum, msg);
          end if
      end for
      EMIT(v, sum);
  DENSESLOT (v, msg):
      v.msg ← VP.mergeMessage(v.msg, msg);
    return v.msg;
```

(c) Push-Pull (Dense Mode)

Fig. 4. Express the workflow of the VCProg programming model (*VP*) with the typical vertex-centric graph programming models.

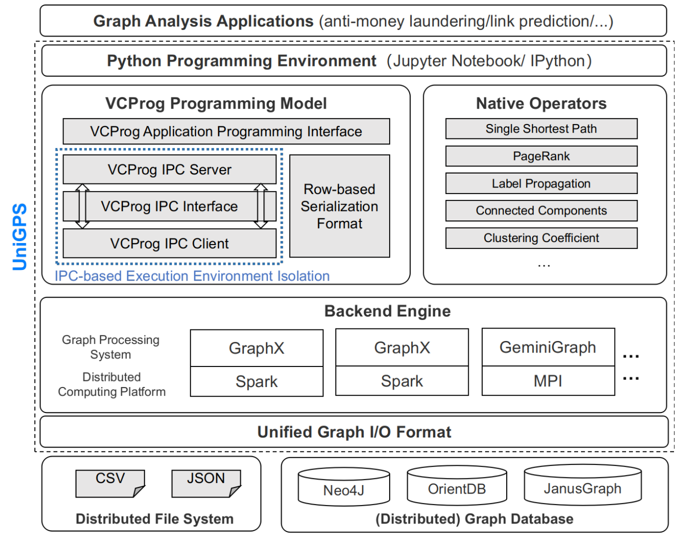

Fig. 5. System architecture of UniGPS.

native operator APIs that allows users calling built-in native implementation of frequently-used processing operators like PageRank and single-source shortest path (SSSP). UniGPS's APIs are platform independent. Programs written with UniGPS can be executed with different distributed graph processing systems by just changing the engine parameter in each API.

B. System Architecture

Fig. 5 shows the system architecture of UniGPS. UniGPS consists of four modules: VCProg programming model, native operators, backend engine, and unified graph I/O format.

The VCProg programming model module provides the VCProg API and runs custom user programs. Since user programs are written in Python, the distributed graph processing systems written in Java and C++ cannot execute them directly. We propose an interprocess-communication-based mechanism to isolate the execution environment of the two parts. We adopt a row-based serialization format to serialize property/message records during communication. We will elaborate on the mechanism later in Section IV-C.

The native operator module contains some frequently-used pre-compiled graph operators. UniGPS natively implements every operator for every distributed graph processing system in advance. UniGPS provides a uniform platform-independent API for each operator. Every API contains an `engine` parameter to select the preferred graph processing system.

The backend engine module integrates VCProg-compatible distributed graph processing systems in UniGPS. Currently, our prototype of UniGPS have integrated Giraph, GraphX, and Gemini. UniGPS can integrate with other systems that are compatible with the VCProg programming model. Each system is regarded as an engine to run VCProg programs and native operators.

The unified graph I/O format module uses a unified graph serialization format (like GraphSON [10]) to decouple the external data sources and distributed graph processing systems.

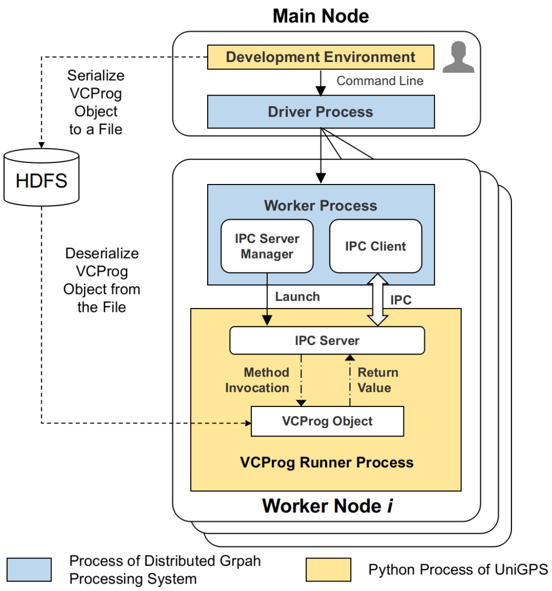

Fig. 6. Execution environment isolation with interprocess communication.

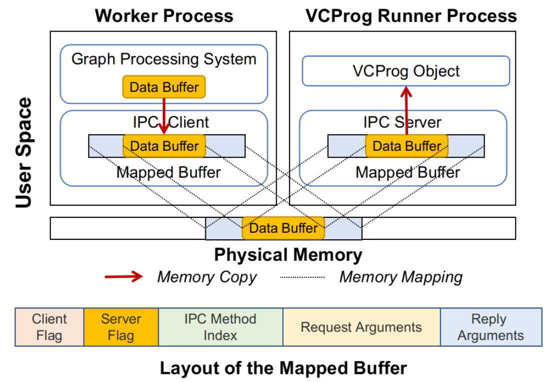

Fig. 7. Memory-mapping-based interprocess communication.

If we want to make M systems read/write N data sources, we have to implement $M * N$ I/O formats without the unified format. Using the unified format as an intermediate format, we only need to implement $M + N$ I/O formats, significantly reducing the development and maintenance costs of UniGPS.

C. Interprocess Communication Based Execution Environment Isolation

The distributed graph processing systems written in Java/Scala/C++ call the user-defined functions via function/method invocation provided by the programming language itself, requiring users developing their programs in the same programming language as the systems.

To enable the existing Java/Scala/C++-based systems to call the user-defined functions (methods) of VCProg written in Python, we propose an execution environment isolation mechanism based on interprocess communication (IPC). As shown in Fig. 5, the mechanism contains three modules: IPC server, IPC interface, and IPC client. The IPC server is a Python process that contains the user-given VCProg object. The IPC client is embedded in the worker processes of distributed graph processing systems. The IPC client and server communicate through the IPC interface. The IPC interface allows the IPC client invoking the methods of the VCProg object in the IPC server with remote procedure call.

1) Workflow of a VCProg Job: With the execution environment isolation mechanism, UniGPS adds several extra steps in the workflow of a VCProg-based graph processing job, as shown in Fig. 6.

Before the job starts, UniGPS serializes the VCProg object that the user submits in the Python environment to a file and uploads it to HDFS.

UniGPS starts a distributed graph processing job based on the system engine specified by the user through command line. The job starts several worker processes among the worker nodes in the cluster. For example, GraphX starts multiple Spark executor processes.

Every worker process launches a dual VCProg runner process. The VCProg runner process deserializes the VCProg object from the file stored on HDFS and creates an IPC server based on the VCProg object. The worker process creates an IPC client and establishes the connection with the IPC server.

After the initialization steps above accomplish, the worker processes start running the processing job according to the workflow of the corresponding distributed graph processing system. For example, the GAS-model-based systems follow the workflow in Fig. 4 to run the job. When the job needs to call a method of the VCProg instance object, it triggers a remote procedure call through the IPC client to the VCProg object in the IPC server.

The IPC-based execution isolation mechanism enables UniGPS to support the Python language for any distributed graph processing system, hide the details of distributed processing from users, and support cross-platform execution of user programs.

2) Interprocess Communication Optimization: Since a UniGPS job triggers interprocess communication (IPC) frequently, the performance of IPC significantly affects the efficiency of UniGPS. Although it can be implemented with general remote procedure call (RPC) frameworks such as gRPC, the RPC frameworks based on the network stack has non-trivial overheads. In those frameworks, data buffers have be copied between the user space and the kernel space multiple times during an remote invocation.

To reduce the overhead of data copies in IPC, UniGPS uses the zero-copy IPC implementation based on the memory-mapped files. UniGPS creates a mapped buffer on both sides of the IPC client and server, as shown in Fig. 7. The two buffers are mapped to the same region in the physical memory through the memory mapping file mechanism provided by Linux. The reading and writing of the mapped buffers directly operates on the corresponding region in the physical memory. Changes to one of the mapped buffers are immediately reflected in the other buffer without any data copy. In other words, the communication between the IPC client and server does not involve any data copy (i.e., *zero copy*) and the communication occurs in the user space, avoiding the overheads of kernel-space switching. With the mapped buffers, the problem of implementing RPC becomes how to organize the concurrent reading and writing of the mapped buffers on the client and server sides.

UniGPS uses the memory layout in Fig. 7 to manage the mapped buffers. The client flag indicates whether the client is ready in preparing the IPC method index and the request arguments. The server flag indicates whether the server finishes the method invocation. The IPC method index indicates which method of the VCProg object the IPC client invokes.

Since RPC invocations usually finishes quickly in VCProg, we use the busy waiting mechanism to synchronize between the IPC client and server. When the client starts a RPC invocation, it first sets the client flag and then repeatedly checks whether the server flag becomes ready. The server side also repeatedly checks the client flag. Once the client flag becomes ready, the server processes the RPC immediately. Compared with the lock-based mechanism, the busy waiting avoids triggering system calls . UniGPS uses the thread yield mechanism in busy waiting to actively give up invalid CPU time slices, reducing the waste in CPU cycles.

Usability Comparison. Table I compares the usability of UniGPS and the existing distributed graph processing systems/frameworks. UniGPS supports the Python language and can be used in an *interactive* development environment like Jupyter Notebook, improving the productivity of program development and debugging. The APIs of UniGPS are platform independent, hiding details of distributed computing from users. UniGPS and KDT are the only two frameworks that satisfy the three usability criteria discussed in Section I-A. However, the customizability of KDT is more limited than UniGPS. KDT only supports using user-defined functions for several semiring methods. UniGPS can customize the methods of all the three phases in each iteration.

V. Performance Evaluation

The section evaluates the scability of UniGPS and compares its computational performance with the stand-alone Python graph processing library NetworkX.

A. Experimental Environment

All experiments were conducted in a cluster with 9 nodes (1 main node + 8 worker nodes) connected via 1Gbps ethernet. Each node was equipped with 8 physical cores, 40 GB memory, and 1.8 TB HDD hard disk. We launched a Ubuntu 20.04 container for each node to provide an isolated experimental environment. We used CPython 3.7.4 as the Python interpreter.

UniGPS integrated three distributed graph processing systems as the backend engines: Giraph [2] (version 1.3.0 with Hadoop 2.7.7 and OpenJDK 1.8), GraphX [3] (with Spark 2.1.2 and Scala 2.11.8), and Gemini [6] (compiled with GCC 9.3.0). For Giraph, UniGPS launched eight workers for each node and allocated 4 GB memory for each worker. For GraphX, UniGPS launched eight Spark executors for each node and allocated 4 GB memory for each executor. For Gemini, each node ran eight single-thread worker processes.

TABLE II
Overview of Real-world Graph Datasets

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	Directed	Source
as-skitter (`as`) [21]	1.7M	22.2M	No	Computer Network
soc-livejournal (`lj`) [21]	4.8M	69.0M	Yes	Social Network
com-orkut (`ok`) [21]	3.1M	234.4M	No	Social Network
uk-2002 (`uk`) [22]	18.5M	298.1M	Yes	WWW

B. Performance Comparison

We compared the execution time of UniGPS with the serial Python graph processing library NetworkX [11] (version 2.5) on four real-world graph datasets (as shown in Table II). Fig. 8a shows the experimental results on several typical graph algorithms. The typical algorithms of UniGPS were implemented with the VCProg API and run on different engines. If the execution time exceeded three hours (i.e., timeout) or the program crashed, the test case was not shown.

The experimental results indicated that UniGPS could handle much larger datasets than NetworkX. NetworkX could not process the large datasets `ok` and `uk` due to the out of memory exception. UniGPS with the Giraph engine could handle all datasets with the help of distributed processing. On the `lj` dataset, the execution time of UniGPS was 21.76% (PageRank), 31.07% (SSSP), and 70.23% (connected components) of NetworkX. UniGPS met timeout with the GraphX and Gemini engines. The edge-parallel design of GraphX and Gemini made the IPC overheads more obvious. UniGPS is more efficient to work with the Giraph engine.

C. Data Scalability

To evaluate the data scalability of UniGPS, we used the logNormalGraph generator of GraphX to generate random graphs of similar topological characteristics but with different scales. Fig. 8b shows the execution time of UniGPS (VCProg API with the Giraph engine) and NetworkX on different scales.

The execution time of UniGPS and NetworkX increased near linearly with the number of edges of the graph. The execution time of UniGPS was much smaller than that of NetworkX. The advantages became clearer as the graph scale increased. It indicates that UniGPS had near-linear data scalability. NetworkX crashed due to the out of memory exception on big graphs. The graph scale that UniGPS could handle was an order of magnitude higher than NetworkX. It indicates that UniGPS can help users process big graphs far beyond the memory capacity of a single machine, in an easy-to-use interactive Python programming environment.

D. Machine Scalability

We measured the execution time of UniGPS (VCProg API with the Giraph engine) with different numbers of CPU cores. Fig. 8c converts the execution time into the speedup relative to the case of 16 cores. The experimental results indicated that

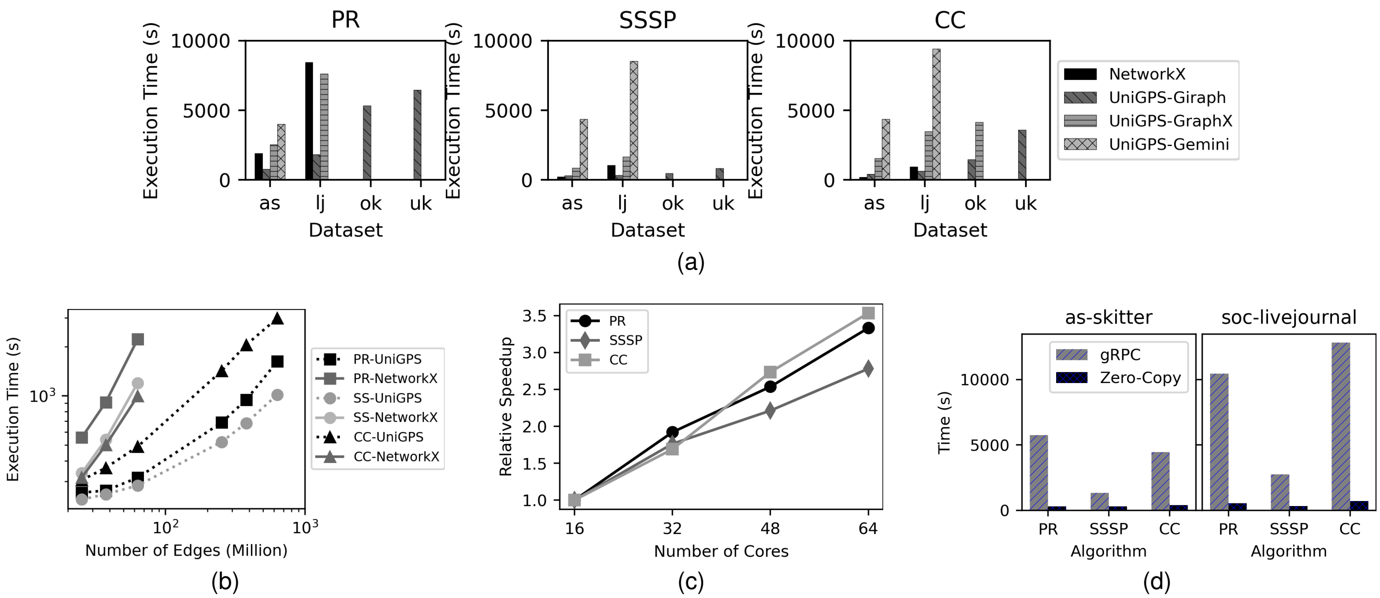

Fig. 8. Performance evaluation of UniGPS on typical graph algorithms PageRank (PR), single-source shortest path (SSSP), and connected components (CC). (a) Performance comparison between UniGPS and NetworkX. (b)Data scalability of UniGPS and NetworkX. (c) Machine scalability of UniGPS. (d) Effects of IPC optimization.

UniGPS shew near-linear machine scalability. The scalability of CC and PR was better than that of SSSP, because the two algorithms were more computationally intensive.

E. Effects of the IPC Optimization

Fig. 8d compares the execution time of UniGPS with two different RPC mechanisms: the network-based gRPC and the proposed zero-copy IPC. The execution time of UniGPS with the zero-copy IPC was much smaller than that of gRPC. gRPC had to copy the data buffers from the user space to the kernel space and trigger several system calls, bringing high overheads. Since UniGPS frequently triggers remote procedure calls during execution, using a zero-copy IPC mechanism can significantly reduce the execution time of UniGPS.

VI. Conclusion and Future Work

The existing distributed graph processing systems are not friendly enough for data analysts and algorithm engineers. To increase the usability of distributed graph processing, we propose a unified graph programming framework UniGPS. UniGPS provides cross-platform unified programming interfaces, hides distributed computing details from users, and supports Python as the programming language. UniGPS adopts a vertex-centric unified graph programming model VCProg compatible with the Pregel, GAS, and Push-Pull model. UniGPS uses an IPC-based execution environment isolation mechanism to enable Java/C++-based systems to call user-defined methods in Python.

In the future, we plan to propose more techniques to improve the performance of UniGPS. One possible technique is to organize the RPC invocation in a pipeline manner to overlap computing and communication. High-performance graph I/O is also an interesting topic to investigate.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China (#U1811461), the National Key R&D Program of China (#2019YFC1711000), the Open Project of State Key Laboratory for Novel Software Technology (#KFKT2021B33), and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Guanghui Zhu and Yihua Huang are corresponding authors with equal contributions.

References

[1] S. Heidari, Y. Simmhan, R. N. Calheiros, and R. Buyya, "Scalable graph processing frameworks: A taxonomy and open challenges," *ACM Computing Surveys*, vol. 51, no. 3, pp. 60:1 – 60:53, 2018.

[2] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, p. 1804–1815, 2015. [Online]. Available: http://giraph.apache.org/

[3] 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 *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, 2014, pp. 599–613. [Online]. Available: https://spark.apache.org/graphx/

[4] S. Cass, "The top programming languages: Our latest rankings put python on top-again - [careers]," *IEEE Spectrum*, vol. 57, no. 8, pp. 22–22, 2020.

[5] G. Piatetsky. (2019) Python leads the 11 top data science, machine learning platforms: Trends and analysis. [Online]. Available: https://www.kdnuggets.com/2019/05/poll-top-data-science-machine-learning-platforms.html

[6] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016, pp. 301–316. [Online]. Available: https://github.com/thu-pacman/GeminiGraph/

[7] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Symposium on Operating Systems*

Design and Implementation (OSDI 12). USENIX Association, 2012, pp. 17–30.
[8] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen, "PowerLyra: Differentiated graph computation and partitioning on skewed graphs," *ACM Transactions on Parallel Computing*, vol. 5, no. 3, pp. 13:1 – 13:39, 2019.
[9] A. Lugowski, D. Alber, A. Buluç, J. R. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis, "A flexible open-source toolbox for scalable complex graph analysis," in *Proceedings of the 2012 SIAM International Conference on Data Mining (SDM)*, 2012, pp. 930–941. [Online]. Available: http://kdt.sourceforge.net/
[10] The Apache Software Foundation. Apache tinkerpop. [Online]. Available: https://tinkerpop.apache.org/
[11] NetworkX: Network analysis in python. [Online]. Available: https://networkx.org
[12] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Computing Surveys*, vol. 48, no. 2, pp. 25:1 – 25:39, 2015.
[13] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Blogel: A block-centric framework for distributed computation on real-world graphs," *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1981–1992, 2014.
[14] A. Quamar, A. Deshpande, and J. Lin, "Nscale: neighborhood-centric large-scale graph analytics in the cloud," *The VLDB Journal*, vol. 23, 125 – 150.
[15] D. Yan, G. Guo, M. M. Rahman Chowdhury, M. Tamer Özsu, W.-S. Ku, and J. C. S. Lui, "G-thinker: A distributed framework for mining subgraphs in a big graph," in *Proceedings of the 2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1369–1380.
[16] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. ACM, 2010, pp. 135–146.
[17] M. A. Rodriguez, "The gremlin graph traversal machine and language (invited talk)," in *Proceedings of the 15th Symposium on Database Programming Languages*. ACM, 2015, pp. 1–10.
[18] V. Batagelj, *Python Packages for Networks*. Springer New York, 2018, pp. 1943–1952.
[19] T. P. Peixoto. graph-tool: Efficent network analysis with python. [Online]. Available: https://graph-tool.skewed.de/
[20] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing," *Proceedings of the VLDB Endowment*, vol. 11, no. 4, pp. 420–431, 2017.
[21] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.
[22] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM, 2004, pp. 595–601. [Online]. Available: http://law.di.unimi.it/datasets.php