RESEARCH-ARTICLE

Nezha: An Efficient Distributed Graph Processing System on Heterogeneous Hardware

PENGJIE CUI, Northeastern University, Shenyang, Liaoning, China

HAOTIAN LIU, Southern University of Science and Technology, Shenzhen, Guangdong, China

DONG JIANG, Northeastern University, Shenyang, Liaoning, China

BO TANG, Southern University of Science and Technology, Shenzhen, Guangdong, China

YE YUAN, Beijing Institute of Technology, Beijing, China

Open Access Support provided by:

Southern University of Science and Technology

Northeastern University

Beijing Institute of Technology

Nezha: An Efficient Distributed Graph Processing System on Heterogeneous Hardware

PENGJIE CUI[*], Northeastern University, China
HAOTIAN LIU, Southern University of Science and Technology, China
DONG JIANG[†], Northeastern University, China
BO TANG[‡], Southern University of Science and Technology, China
YE YUAN[‡], Beijing Institute of Technology, China

The growing scale of graph data across various applications demands efficient distributed graph processing systems. Despite the widespread use of the *Scatter-Gather* model for large-scale graph processing across distributed machines, the performance still can be significantly improved as the computation ability of each machine is not fully utilized and the communication costs during graph processing are expensive in the distributed environment. In this work, we propose a novel and efficient distributed graph processing system Nezha on heterogeneous hardware, where each machine is equipped with both CPU and GPU processors and all these machines in the distributed cluster are interconnected via Remote Direct Memory Access (RDMA). To reduce the communication costs, we devise an effective communication mode with a graph-friendly communication protocol in the graph-based RDMA communication adapter of Nezha. To improve the computation efficiency, we propose a multi-device cooperative execution mechanism in Nezha, which fully utilizes the CPU and GPU processors of each machine in the distributed cluster. We also alleviate the workload imbalance issue at inter-machine and intra-machine levels via the proposed workload balancer in Nezha. We conduct extensive experiments by running 4 widely-used graph algorithms on 5 graph datasets to demonstrate the superiority of Nezha over existing systems.

CCS Concepts: • **Information systems → Graph-based database models**; • **Theory of computation →** *Graph algorithms analysis.*

Additional Key Words and Phrases: Graph Processing, Heterogeneous Hardware

ACM Reference Format:
Pengjie Cui, Haotian Liu, Dong Jiang, Bo Tang, and Ye Yuan. 2025. Nezha: An Efficient Distributed Graph Processing System on Heterogeneous Hardware. *Proc. ACM Manag. Data* 3, 1 (SIGMOD), Article 57 (February 2025), 27 pages. https://doi.org/10.1145/3709707

1 Introduction

Graph is a well-known data structure used to represent the relationships among the entities in various applications, including social network analysis [6, 8], biological data interpretation [39],

[*]This work was done when Pengjie Cui was a research assistant at Southern University of Science and Technology.
[†]This work was done when Dong Jiang was a research assistant at Southern University of Science and Technology.
[‡]Both Bo Tang and Ye Yuan are corresponding authors.

Authors' Contact Information: Pengjie Cui, Northeastern University, Shenyang, China, 1810602@stu.neu.edu.cn; Haotian Liu, Southern University of Science and Technology, Shenzhen, China, 12231144@mail.sustech.edu.cn; Dong Jiang, Northeastern University, Shenyang, China, jiangdongcs@stumail.neu.edu.cn; Bo Tang, Southern University of Science and Technology, Shenzhen, China, tangb3@sustech.edu.cn; Ye Yuan, Beijing Institute of Technology, Beijing, China, yuan-ye@bit.edu.cn.

ACM 2836-6573/2025/2-ART57
https://doi.org/10.1145/3709707

and web graph mining [64]. With the recent development in information science and big data applications, graph data has grown rapidly. For example, the social graph of Facebook has reached terabyte-scale [6], which obviously exceeds the capacity of a single machine. As a result, many studies [13, 21, 31, 42, 58, 68, 71] have turned to process the large-scale graph data using distributed clusters. For example, Gemini [71] is built on a CPU cluster to achieve high scalability and efficiency during large-scale graph processing, and Lux [17] uses multiple GPUs across different machines to improve the throughput of graph processing. However, existing techniques in these systems are insufficient to build an efficient distributed graph processing system on heterogeneous hardware, where each machine is equipped with both CPU and GPU processors and all these machines are interconnected via RDMA. We next elaborate on the limitations of existing techniques from two aspects: (i) computation efficiency and (ii) communication costs.

Computation Efficiency. Both Gemini [71] and Lux [17] utilize homogeneous computing processors in the distributed environment. In particular, Gemini [71] is built upon a CPU cluster, and Lux [17] is a distributed graph processing system on GPUs, with its CPUs playing a supportive role. However, modern machines in a cluster are typically equipped with both CPU and GPU processors. It is obvious that existing techniques in both Gemini [71] and Lux [17] cannot fully utilize their computational power. Besides, many graph processing systems [7, 12, 69] have been proposed to cooperate with both CPU and GPU in a single machine for efficient graph processing. Unfortunately, these single machine solutions ignore the communication costs in the distributed cluster. Thus, they cannot be directly adapted to build an efficient distributed graph processing system on heterogeneous hardware.

Moreover, improving computation efficiency by balancing workloads across different machines is not trivial in a distributed graph processing system. Graph partitioning is a common idea to alleviate the workload imbalance issue. Various partition-based methods have been proposed in the literature, such as vertex-cut [13, 15, 32, 33, 38], edge-cut [49, 55, 71], and hybrid-cut [5]. However, the effectiveness of a graph partitioning solution highly depends on the raw graph data and upstream applications. Therefore, it is insufficient to improve the distributed graph processing system by only using graph partitioning approaches [36]. Mizan [21] proposed a solution to dynamically balance the workload in the cluster via monitoring techniques. However, it introduces overhead to monitor and adjust the workload placement, which can be larger than the benefits gained [29]. Thus, these techniques cannot be directly used to build an efficient distributed graph processing system.

Communication Costs. Besides the computation efficiency, the end-to-end distributed graph processing performance also depends on communication costs. Distributed graph processing is communication intensive [1, 5, 13], as it requires transferring large amounts of data among machines to synchronize intermediate results, which incurs significant communication costs. Most existing distributed graph processing systems [21, 31, 42], including Gemini [71] and Lux [17], utilize the TCP/IP protocol for communication. However, the high CPU overhead and multiple data copy costs associated with TCP/IP have severely impacted the performance of distributed graph processing. To reduce communication costs, some existing distributed graph processing systems [15, 33, 71] have implemented specific software-oriented optimization strategies, such as Gemini [71] assigns dedicated communication threads to hide the TCP/IP costs. Unfortunately, high CPU invocation and data copy costs cannot be fully avoided through software-oriented optimizations [66].

Recently, Remote Direct Memory Access (RDMA) has been widely used in distributed systems [2, 9, 27, 73], which allows direct memory access between machines without involving the CPU. For example, FaRM [9] introduces an RDMA communication stack designed for efficient transaction processing. Wukong+G [62] uses RDMA to accelerate its Resource Description Framework (RDF). However, existing RDMA communication methods do not consider the characteristics of distributed graph processing. For instance, Wukong+G initiates communication for each individual request. This approach leads to one RDMA communication per vertex and introduces unnecessary overhead for graph processing. Furthermore, these RDMA communication methods overlook the issue of memory efficiency, which is critical for efficiently processing large-scale graphs in a distributed graph processing system.

In this work, we propose Nezha, an efficient distributed graph processing system on heterogeneous hardware (i.e., CPU, GPU, and RDMA). Firstly, a novel graph-based RDMA communication adapter is introduced in Nezha. In particular, it includes a graph-friendly RDMA connection mode and a graph-based RDMA communication protocol, which significantly reduce the communication costs and improve the memory utilization during the distributed graph processing. Secondly, a multi-device cooperative executor is devised in Nezha, which improves end-to-end graph processing performance via a thoroughly designed multi-device cooperative execution model. Moreover, it improves the end-to-end performance by overlapping PCIe costs, CPU/GPU computation costs, and RDMA communication costs during graph processing. Thirdly, an online workload balancer is designed in Nezha, which balances the workload in the distributed cluster at both inter-machine level and intra-machine level. To sum up, the technical contributions of this work can be summarized as follows.

- **The Graph Processing System: Nezha**. We propose Nezha, to the best of our knowledge, it is the first graph processing system which utilizes both CPU and GPU processors on the distributed machines interconnected via RDMA. The performance of Nezha is the best among all existing distributed graph processing systems as it not only improves the computation efficiency but also reduces the communication costs.
- **A Novel Graph-Based RDMA Communication Mechanism**. We devise a novel graph-based RDMA communication adapter in Nezha. In particular, we propose a graph-friendly RDMA connection mode to reduce the memory usage during graph processing. Moreover, we devise a graph-based RDMA communication protocol to reduce the communication costs in the distributed system by improving the message buffer utilization, synchronizing the sender and receiver threads efficiently. Besides, the above protocol is workload adaptive as it introduces a congestion control method for various workloads.
- **A Multi-Device Cooperative Execution Model**. We propose a multi-device cooperative executor in Nezha, which improves end-to-end performance via the CPU-GPU cooperative execution solution in the distributed cluster. In addition, we further optimize the performance by overlapping (i) PCIe data movement and GPU computation costs with GPU steaming techniques, and (ii) RDMA communication costs and CPU computation costs via a threshold-triggered *Scatter-Gather* model.
- **Intra- and Inter- Machine Workload Balance Method**. We design an effective workload balancer that optimizes workload distribution both within and across machines in Nezha. Specifically, we propose an interrupt-free task stealing mechanism that utilizes the kernel bypass advantages of one-sided RDMA *READ* for inter-machine workload balancing.

We conduct comprehensive experiments to verify the superiority of Nezha by comparing with three state-of-the-art existing distributed graph processing systems. Specifically, we present the minimum and maximum speedup of Nezha over these three competitors in Table 1 by running four widely used graph algorithms: Breadth-First Search (BFS), Single-Source Shortest Path (SSSP), Weakly Connected Components (WCC), and PageRank (PR) on five graph datasets. The '—' indicates the case that all other systems cannot process the large-scale graph. Nezha achieves up to 4.54X,

Table 1. The speedup times of Nezha

Dataset	BFS	SSSP	WCC	PR
friendster	1.27~2.31X	2.63~4.40X	1.18~2.22X	2.06~4.48X
un-union	1.88~3.25X	1.79~4.13X	1.45~3.32X	1.27~3.27X
rmat-53	2.51~4.54X	4.21~6.24X	2.34~4.80X	2.89~5.99X
clubeweb12	2.47~3.56X	3.94~—X	2.31~2.86X	3.11~4.61X
rmat-127	2.11~—X	—	3.14~—X	3.32~—X

6.24X, 4.80X and 5.99X speedup over them by processing BFS, SSSP, WCC and PR on tested graph datasets, respectively.

The rest of the paper is organized as follows. We introduce the preliminaries of distributed graph processing system in Section 2. Then, we propose the novel distributed graph processing system Nezha in Section 3. Next, we elaborate on the detailed techniques of our RDMA communication adapter, multi-device cooperative executor and inter-/intra-machine workload balancer of Nezha in Sections 4, 5 and 6, respectively. We conduct experimental evaluation in Section 7 and discuss the related work in Section 8. Finally, we conclude the paper in Section 9.

2 Preliminaries

In this section, we introduce the preliminaries of distributed graph processing system, which includes processing procedure, computation model and execution paradigm.

Distributed Graph Processing Procedure. It consists of two steps: (i) graph partitioning and (ii) graph computing. The graph partitioning step divides the entire large graph into small subgraphs, and assigns them to the machines in a distributed cluster. In the graph computing step, every machine in the distributed cluster processes the assigned subgraph and communicates with others to synchronize the intermediate results. As discussed in the Introduction, partitioning the graph alone is insufficient to achieve optimal performance in a distributed graph processing system [36]. In this work, we focus on the graph computing step, which is orthogonal to graph partitioning solutions. We next elaborate the computation model in distributed graph processing system.

Distributed Computation Model. It abstracts the algorithmic logic of graph processing algorithms and provides simple and effective interfaces for end-users. The *Scatter-Gather* computation model [50] is widely used in existing distributed graph processing systems [46, 50, 67]. In particular, the *Scatter* propagates the state of updated vertices from the local machine to other machines, while the *Gather* updates vertex states based on the received messages. Many graph processing algorithms can be easily implemented using the *Scatter-Gather* computation model in distributed graph processing systems [19].

Execution Paradigm. The execution paradigms of existing graph processing systems can be classified into three categories by considering their execution units: (i) vertex-centric [5, 13, 16, 17, 31, 56, 58, 60, 63, 71], (ii) edge-centric [24, 41], and (iii) subgraph-centric [10, 48, 53, 61, 68].

3 Our Proposal Nezha

In this section, we introduce Nezha, a novel distributed graph processing system designed to provide excellent graph processing performance on distributed machines with heterogeneous hardware (i.e., CPU, GPU, and RDMA). We present the system overview of Nezha in Section 3.1, and introduce its major programming interfaces in Section 3.2.

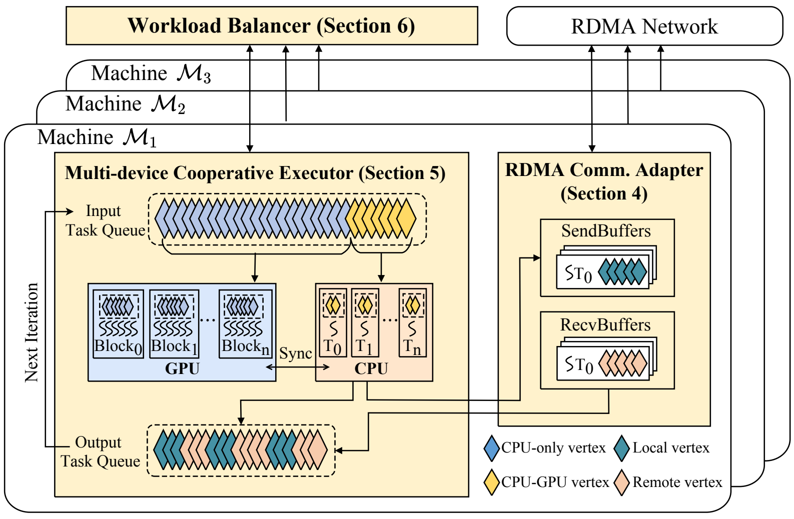


Fig. 1. System Overview of Nezha

3.1 System Overview of Nezha

Figure 1 depicts the overview of our proposed Nezha. In particular, we adapt the graph partitioning method from Gemini [71] to distribute the subgraphs across machines in a cluster. For graph computing, Nezha utilizes *Scatter-Gather* computation model and the vertex-centric paradigm by following existing distributed graph processing systems [31, 43, 50]. The excellent distributed graph processing performance offered by Nezha owes to three key components in it.

- **RDMA Communication Adapter.** We introduce a new graph-based RDMA communication protocol in Nezha, which controls the send buffer and the receive buffer of each thread and manages the communication across machines in a graph-friendly manner.
- **Multi-device Cooperative Executor.** To improve the utilization of heterogeneous hardware (e.g., CPU, GPU, and RDMA) in the distributed cluster, we propose a multi-device cooperative execution model in Nezha, which accelerates the graph processing by fully utilizing the computational power of the CPU and GPU processors and significantly overlapping the computation costs and communication costs during distributed graph processing.
- **Intra- and Inter- Machine Workload Balancer.** To alleviate the impact of workload imbalance on the distributed graph processing system, we devise an intra- and inter- machine workload balance scheme in Nezha, which balances the workload among the distributed machines efficiently and effectively.

In subsequence, for each component in Nezha, we will first introduce the limitations of existing techniques, then present our proposed solution and highlight its technical contributions.

3.2 Programming Interface of Nezha

Nezha utilizes the *Scatter-Gather* model [50] as in existing studies [31, 43, 50], which is easy to implement various graph processing algorithms by the three key programming interfaces.

- sendMessage: It packages the vertex ID and vertex state into a message and sends it to the target machine. It automatically determines the target machine of the vertex via the graph partitioning solution.

Algorithm 1: BFS Implementation on Nezha

```
1  $L \leftarrow \{+\infty, +\infty, +\infty, ..., +\infty\}$; // A set for vertex level
2  $L[root] \leftarrow 0$; // Set root level to zero
3  $A_{curr} \leftarrow root$;
4  Scatter(source, dest) begin
5      $level \leftarrow L[source.id] + 1$ ;
6      if $L[dest.id] > level$ then
7          $sendMessage(dest.id, L[dest.id])$ ;
8  Gather(msgs) begin
9      foreach $msg$ in $msgs$ do
10         if $L[msg.id] > msg.state$ then
11             $L[msg.id] \leftarrow msg.state$ ;
12             $A_{next} \leftarrow A_{next} \cup msg.id$ ;
13 BFS begin
14     while $A_{curr} \neq \emptyset$ do
15         $A_{next} \leftarrow \emptyset$;
16         $A_{next} \leftarrow processEdge$ ($A_{curr}$, Scatter, Gather);
17         $A_{curr} \leftarrow A_{next}$;
```

- processEdge: It applies the algorithm-specific *Scatter* function to those edges whose source vertices are in the active vertex set, and applies the algorithm-specific *Gather* function to update the vertex state with the received messages. It returns an updated vertex set for the next iteration during graph processing.
- processVertex: Similar to processEdge, it applies the algorithm-specific vertex updating function to all vertices in the active set, and returns an updated vertex set for the next iteration.

We next demonstrate the ease-of-use property of Nezha by implementing Breadth First Search (BFS) algorithm via these programming interfaces. The pseudocode of BFS is illustrated in Algorithm 1. In the first three lines, a set L is used to store the level of all vertices, where the level of the root vertex is initialized to 0 and the others are set to $+\infty$. The active set A_{curr} includes all active vertices in this iteration. The new active vertices will be stored in A_{next}, which will be used in the next iteration. It iteratively invokes *processEdge* until the levels between every vertex to the root are computed. The *Scatter* (Lines 4-7) propagates the updated value of the destination vertex via the sendMessage interface when needed (Lines 6-7). The *Gather* updates the vertex state with the received messages (Lines 8-12), and if the vertex state is updated in the received messages, it will be activated in the next iteration. As we will show in experimental evaluation, other graph algorithms can also be implemented easily on Nezha via the three provided programming interfaces. For example, processVertex will be used in the PageRank to update the vertex states accordingly.

In order to cooperatively process the graph algorithms in both CPUs and GPUs in Nezha, we need to provide both CPU and GPU implementations for algorithm-specific computation functions, e.g., *Scatter* in Algorithm 1. However, implementing the GPU version of these functions is trivial as it shares the same algorithmic logic with its CPU version and only needs to introduce GPU-specific operations (e.g., atomic operators).

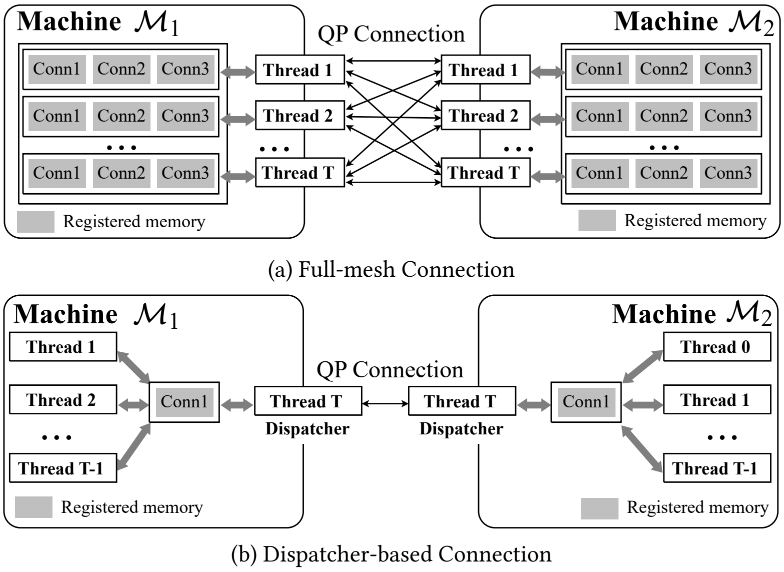


Fig. 2. Two typical RDMA connection modes

4 Graph-based RDMA Communication Adapter

In a distributed system, it is crucial to offer an efficient communication mechanism among the machines in a cluster. To achieve this, we propose a novel graph-based RDMA communication adapter in this section. Specifically, we first introduce the RDMA connection mode in Section 4.1, then elaborate on the research challenges to build an efficient RDMA communication adapter in Section 4.2, and finally devise the graph-friendly RDMA communication mechanism in Section 4.3.

4.1 RDMA Communication Mode

We first introduce the basic concepts of RDMA and then present the RDMA connection modes used in existing distributed systems.

RDMA Network. The RDMA-connected machines communicate with others via queue pairs (QPs). Each QP consists of a send queue and a receive queue. When an application initiates an RDMA operation to send or receive data, it will post a work queue element (WQE) to the send queue or the receive queue using RDMA *verbs*. In particular, RDMA network interface card (RNIC) provides two types of *verbs*: *one-sided verbs* and *two-sided verbs*. RDMA *READ* and RDMA *WRITE* are *one-sided verbs*, which access the remote memory without involving the remote CPU. RDMA *SEND* and RDMA *RECEIVE* are *two-sided verbs*, which are similar to traditional Linux sockets. Existing studies have shown that one-sided operations are more efficient and scalable than two-sided operations [4, 9, 51].

RDMA Connection Modes in Distributed System. RDMA offers three transport types: Reliable Connected (RC), Unreliable Connected (UC), and Unreliable Datagram (UD). RC is the most widely used transport type in various distributed systems [9, 58] because it is reliable and supports all *one-sided verbs*. However, it requires one-to-one connections between QPs. We next introduce two widely used RDMA connection modes for the RC.

Full-mesh connection: It is the most straightforward connection mode in RDMA, and it has been used in various distributed systems, such as the RDMA-based MXNet framework [26]. In particular, each pair of threads on two machines creates a QP connection so that any thread in a

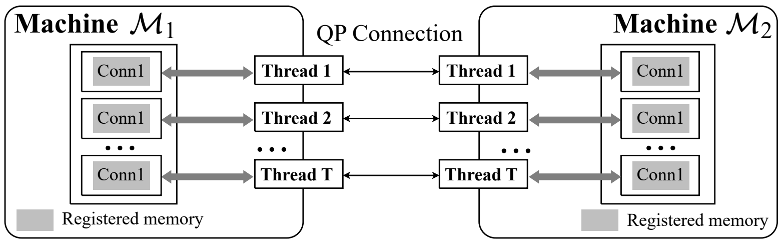


Fig. 3. Our proposed RDMA connection mode

machine can independently communicate with every thread in the other machine, see Figure 2(a). The kernel-bypass property of RDMA requires sufficient memory to be registered for each QP connection. Therefore, the required RDMA memory for each machine depends on the number of QP connections it has. In full-mesh connection mode, the number of QP connections is $(M-1) \times T^2$, where M is the number of machines in a distributed cluster and T is the number of threads in each machine. Obviously, the full-mesh connection mode incurs a large amount of memory consumption. For example, suppose a cluster has 6 machines and each machine has 10 threads, it consumes 62.5 GB of memory for RDMA in each machine if a QP connection allocates a 128MB memory buffer. Moreover, RNIC faces the cache thrashing problem [20] with the increasing number of QP connections as they cache various types of information, e.g., connection-related states.

Dispatcher-based connection: To overcome the limitations of full-mesh connection mode, dispatcher-based connection mode [58] has been proposed, see Figure 2(b). Specifically, it introduces one or more communication dispatcher(s) to manage the communication requests from all threads on the same machine. Compared to full-mesh connection, the number of QP connections is reduced to $(M - 1) \times D^2$, where M is the number of machines, and D is the number of dispatchers per machine. It significantly reduces the number of QP connections, since the number of dispatchers is much smaller than the number of threads in each machine. However, the dispatchers in each machine can become performance bottlenecks due to severe concurrency contention in communication-intensive applications [72].

4.2 Research Challenges

With the above analysis of existing RDMA connection modes, we then highlight the research challenges to build an efficient RDMA communication adapter in Nezha.

C4.1: Effective RDMA Connection Mode. Since the full-mesh connection incurs significant memory consumption and cache thrashing issues, and the dispatcher-based connection introduces extra CPU overhead from using dispatchers. Thus, the first research challenge is to devise an effective RDMA connection mode that ensures high memory efficiency for graph-based RDMA communication adapter.

C4.2: Efficient Communication Protocol for Graph Processing. The memory consumption of graph processing is inherently high. For example, running PageRank [54] with 1.3GB graph data on GraphX [14] and Spark [65] uses 12GB and 16GB memory heap, respectively. The distributed graph processing system [71] uses the maximum pre-allocated memory to avoid buffer overflow. Thus, how to design a graph-based communication protocol that achieves excellent memory utilization is the second research challenge for graph-based RDMA communication adapter.

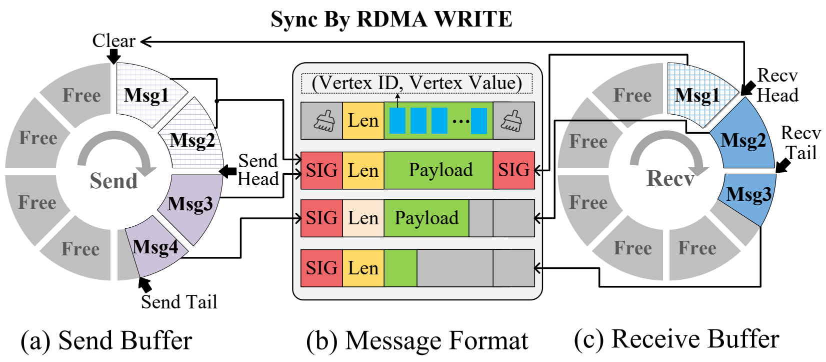


Fig. 4. The send buffer, message format and receive buffer

4.3 Communication Mechanism in Nezha

In this section, we introduce the RDMA communication adapter in Nezha, which addresses the above two challenges.

The Proposed Connection Mode in Nezha. The full-mesh connection enables independent communication among threads on different machines, which is useful for thread-oriented tasks. For example, each thread in one machine wants to communicate with any thread in other machines. However, distributed graph processing is machine-oriented where the graph data is partitioned across different machines, and the intermediate results only need to be synchronized at the machine-level. Returning to the BFS example in Algorithm 1, it only needs to guarantee that the updating messages are correctly sent to the corresponding machines and the new messages are updated on the local machine.

With the above useful observation, we propose the RAMA connection mode in Nezha, as shown in Figure 3. The principle of the connection establishment in Nezha is to minimize the number of QP connections while guaranteeing each thread can send messages to all other machines independently. We achieve this by connecting threads with the same ID across all machine pairs. This reduces the number of QP connections to $(M - 1) \times T$ in each machine. For example, the number of QP connections will be 50 if there are 6 machines in a cluster and each machine has 10 threads. Obviously, the dedicated memory required by this connection mode will be less than the full-mesh connection. Moreover, it is better than the dispatcher-based connection since it does not require a dedicated thread for communication.

With the proposed connection mode, the RDMA communication protocol can be easily implemented as follows: (1) each thread initializes sufficient send and receive buffers; (2) each thread transmits basic connection information to the corresponding threads on other machines to establish connections; and (3) RDMA one-sided *verbs*, i.e., RDMA *WRITE* and *READ*, are invoked to send and receive messages in the distributed cluster.

Graph-based RDMA Communication Protocol. To further improve the RDMA communication efficiency for distributed graph processing in Nezha, we design a novel graph-based RDMA communication protocol based on the proposed RDMA connection mode.

For each QP connection, a message buffer needs to be allocated in memory on both the sender and receiver machines to store the communicating messages. There are three existing buffer management methods: (i) dynamic buffer allocation [25], (ii) persistent buffer association [9, 28, 58, 59], and (iii) hybrid method [30]. Due to the frequent buffer allocation and deallocation, methods (i) and (iii) introduce significant overhead [11]. Therefore, our proposed communication protocol is

Table 2. Summary of our proposed communication protocol

Protocol	Memory Efficiency	Synchronization Cost	Workload-adaptive
Persistent Buffer Association	Low	Expensive	No
Our graph-based protocol	**High**	**Cheap**	**Yes**

based on persistent buffer association. However, there are three major limitations when directly adapting persistent buffer association in the graph-based RDMA communication adapter. Firstly, its memory usage is inefficient since it lacks an efficient reuse strategy. Secondly, the synchronization between the receiver and sender is expensive, e.g., using auxiliary verbs for timely or periodic synchronization. Last but not least, the communication requirements vary significantly among different machines in distributed graph processing. Therefore, adaptive adjustment of communication frequency is necessary. However, existing synchronization schemes do not support this.

To overcome these limitations, we propose a graph-based RDMA communication protocol in Nezha. In particular, two fixed-size message buffers are pre-allocated for the sender and receiver of each QP connection in Nezha, and both buffers are organized as rings to enable more efficient memory usage. As shown in Figure 4(a) and 4(c), the send ring buffer has three pointers, i.e., *Send Head* pointer, *Send Tail* pointer, and *Clear* pointer. In particular, the *Send Head* points to the address of the message in the send buffer which will be sent to the receiver (e.g., Msg 3 and Msg 4). The *Send Tail* shows the address of the send buffer which will be written for the next message. The special-designed *Clear* pointer is used to synchronize the processing progress between the sender and receiver, which can be used to improve the memory utilization of the ring buffer. If the *Send Tail* pointer is close to *Clear* pointer, it indicates that the send buffer is almost full. We use two pointers to manage the receive buffer, i.e., *Recv Head* and *Recv Tail.* The *Recv Head* points to the position of the last message that has been fully processed, its address will be synchronized to the *Clear* pointer in send buffer on-demand, e.g., The *Recv Head* synchronizes with the send buffer when the receiver processed a certain number of updates in the buffer, (e.g., 40% messages in the receive buffer). The *Recv Tail* points to the last message that has been received but it is still waiting for processing.

The message format in our protocol is shown in Figure 4(b). Specifically, each message includes four items: (i) the beginning signature, (ii) message length, (iii) payload, and (iv) the ending signature. For efficient communication in Nezha, we pack a set of vertices in a message and send it by one RDMA *WRITE.* With this format, the receiver can independently determine whether it has received a valid message by examining the beginning and ending signatures of the message in receive buffer.

To support adaptive workload communication in distributed graph processing, we propose a congestion control method based on the above graph-based communication protocol. Specifically, when the *Send Tail* in send buffer catches up the *Clear* pointer, the sending process will be blocked to avoid overwriting. To avoid this situation, the sender changes the normal signature of the message to 'hurried' signature when the *Send Tail* is close to *Clear* pointer. The threshold for this 'hurried' signature can be configured, and we will show its effect on performance in Section 7.3.4. Once the receiver receives a message with the 'hurried' signature, it will fetch and process the message more frequently. Additionally, the 'hurried' signature alleviates the congestion of the send buffer by prompting the receiver to synchronize its *Recv Head* with *Clear* pointer in the sender. We will show the congestion control method during graph processing in Section 5. Table 2 summarizes the advantages of our proposed graph-based communication protocol in RDMA-communication adapter of Nezha.

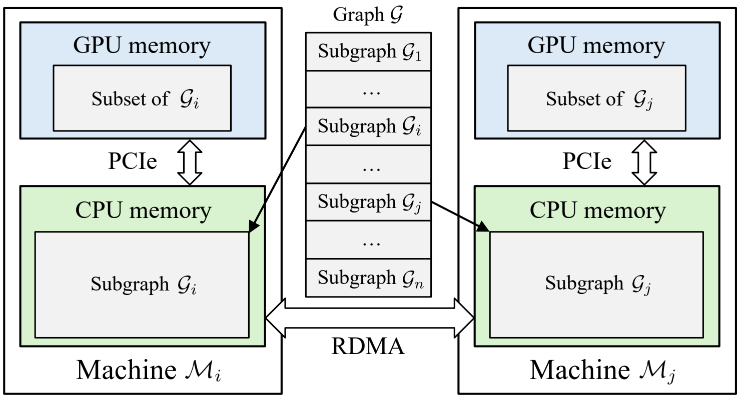


Fig. 5. Graph partition and subgraph extraction in Nezha

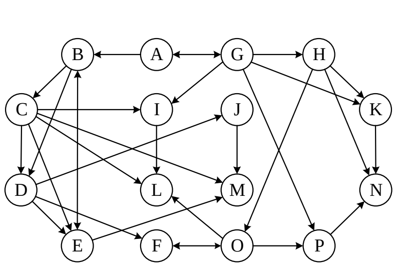
5 Multi-Device Cooperative Executor

Architecting an efficient multi-device cooperative executor on Nezha is still not trivial even with the proposed RDMA communication adapter in Section 4. The core reason is that existing work does not consider the heterogeneous CPU-GPU processors in a distributed environment. For example, existing distributed GPU graph processing system [17] uses CPUs as a supportive role and ignore their computation power, and existing single-machine CPU-GPU graph processing system [7] cannot be directly adapted to a distributed cluster as it do not consider communication across machines. In this section, we first introduce the distributed graph processing procedure on Nezha in Section 5.1, and then we propose performance optimization techniques to build an efficient multi-device cooperative executor in Section 5.2.

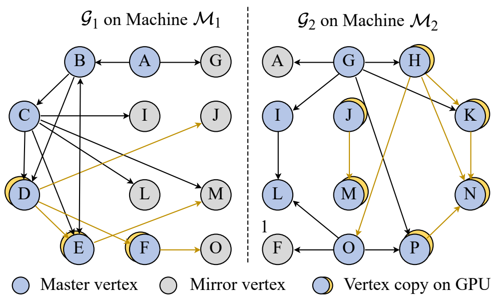
5.1 Distributed Graph Processing on Nezha

Before we introduce the distributed graph processing procedure in Nezha, we first present the graph allocation method in a distributed cluster where each machine has heterogeneous CPU-GPU processors. Figure 5 depicts the allocated subgraph in any two machines $\mathcal{M}_i$ and $\mathcal{M}_j$ of Nezha. Nezha first partitions the graph into a set of subgraphs and assigns them to the CPU memory on each machine by following the method used in existing distributed graph processing systems [5, 13, 17, 21, 31, 58, 71]. Specifically, Nezha uses the same graph partitioning approach of Gemini [71]. For example, the subgraphs $\mathcal{G}_i$ and $\mathcal{G}_j$ are assigned to machines $\mathcal{M}_i$ and $\mathcal{M}_j$, as shown in Figure 5. Next, for every machine $\mathcal{M}_i$, Nezha extracts a subset of $\mathcal{G}_i$ and moves it to the GPU memory of this machine. We use the subgraph extraction algorithm in a single-machine CPU-GPU co-processing system [7] as the subset of $\mathcal{G}_i$ is loaded to the GPU once but can be utilized among multiple iterations. The entire raw graph is stored on the disk of each machine.

We then present the distributed graph processing procedure in Nezha by running the Breadth-First Search (BFS) algorithm on a tiny cluster with two machines. The input graph and partitioned subgraphs are shown in Figures 6(a) and (b), respectively. Specifically, the vertex set of subgraph $\mathcal{G}_1$ on $\mathcal{M}_1$ is {A, B, C, D, E, F} and the vertex set of $\mathcal{G}_2$ on $\mathcal{M}_2$ is {G, H, I, J, K, L, M, N, O, P}. Without loss of generality [13], the vertices in the local machine are master vertices, while those in the remote neighbors are mirror vertices. For example, the gray vertices $\{G, I, J, L, M, O\}$ of $\mathcal{G}_1$ in Figure 6(b) are mirror vertices of machine $\mathcal{M}_1$. For each machine, a subset of master vertices will be extracted and transferred to its GPU memory, e.g., the vertices $\{D, E, F\}$ of $\mathcal{G}_1$ are transferred to its GPU, as shown in Figure 6(b).



(a) Original Graph $\mathcal{G}$



$\mathcal{G}_1$ on Machine $\mathcal{M}_1$
$\mathcal{G}_2$ on Machine $\mathcal{M}_2$
Master vertex
Mirror vertex
Vertex copy on GPU

(b) Subgraphs $\mathcal{G}_1$ and $\mathcal{G}_2$ on $\mathcal{M}_1$ and $\mathcal{M}_2$

Fig. 6. An illustrative example of distributed graph processing on Nezha

Next, we will present the distributed graph processing procedure on Nezha by running BFS on graph $\mathcal{G}$ with starting vertex A. Tables 3(a) and (b) show the value of each vertex of subgraphs $\mathcal{G}_1$ and $\mathcal{G}_2$ on $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively. In particular, we use a triplet [CPU, GPU, Remote] for each vertex to maintain the value which is updated by the CPU, GPU, and remote machine. '∞' is the initial value of each vertex and '-' indicates that the vertex cannot be updated by the GPU or remote machines. For example, the initial value of vertex G is [∞,-,-] as it only can be updated by the CPU in $\mathcal{M}_1$. However, the initial values of vertex F is [∞, ∞, ∞] as it can be updated by any of the CPU, GPU in $\mathcal{M}_1$ or remote machine. We use a single number to show the final value of that vertex, e.g., the value of A is 0 in $\mathcal{M}_1$ as it is the root vertex, and the final value of vertex B is 1 in machine $\mathcal{M}_1$, and it will not be updated after the 1st iteration. The graph processing terminates when all master vertices in every machine are not updated anymore, and the final values of them are the result of BFS.

1st iteration. The outgoing edges of vertex A are computed by the CPU in $\mathcal{M}_1$, thus, the CPU values of vertex G and B are updated to 1, see red color values at 1st iteration in Table 3(a). Since G is a mirror vertex in $\mathcal{M}_1$, its updated value will be sent to $\mathcal{M}_2$ as G is the master vertex in it, see red value at 1st iteration in Table 3(b). The vertex B in $\mathcal{M}_1$ and vertex G in $\mathcal{M}_2$ are the active vertices for the next iteration.

2nd iteration. The outgoing edges of vertex B are computed by CPU, and the values of C, D, and E are updated to 2 in $\mathcal{M}_1$, see 2nd iteration in Table 3(a). Meanwhile, $\mathcal{M}_2$ updates the values of vertices A, H, K, P and I to 2 by CPU, see 2nd iteration in Table 3(b). The updated value of A in $\mathcal{M}_2$ (i.e., 2) will be sent to $\mathcal{M}_1$ since A is a mirror vertex in $\mathcal{M}_2$. However, it does not change the value of A in $\mathcal{M}_1$ as the maintained value of A is less than 2.

3rd iteration. The vertices M, I and L are updated by the CPU in $\mathcal{M}_1$, and vertex L is updated by the CPU in $\mathcal{M}_2$ in this iteration. Moreover, the neighbors of D and E in $\mathcal{M}_1$ are updated by GPU as both D and E are in GPU memory, i.e., the GPU values of vertices F, J and M are 3 in $\mathcal{M}_1$. Similarly, the values of vertices N and O are updated by the GPU in $\mathcal{M}_2$, which are 3 in this iteration. After that, the updated values in the GPU will be synchronized to the CPU in the same machine. Lastly, $\mathcal{M}_1$ sends the updated values of vertices J, M, I and L to $\mathcal{M}_2$.

4th iteration. Following the same procedure, vertex O is updated to 4 in $\mathcal{M}_1$, and vertex F is updated to 4 in $\mathcal{M}_2$. Both O and F will be sent to the corresponding machines, as the updates are from the machines that have their mirror vertices. However, the values of O and F will not be updated, as the values of their master vertices are smaller than updates.

Until now, the BFS terminates and the final values of each vertex in graph $\mathcal{G}$ are the value of every master vertex in Table 3.

Table 3. The illustration of running BFS on Nezha with root vertex A

Iteration	D	E	F	A	B	C	G	J	M	O	I	L
	CPU/GPU	CPU/GPU	CPU/GPU	CPU	CPU	CPU	Mirror vertices					
Initialization	[∞,∞,-]	[∞,∞,-]	[∞,∞,∞]	0	[∞,-,-]	[∞,-,-]	[∞,-,-]	[∞,∞,-]	[∞,∞,-]	[∞,∞,-]	[∞,-,-]	[∞,-,-]
Iteration 1	[∞,∞,-]	[∞,∞,-]	[∞,∞,∞]	0	[1,-,-]	[∞,-,-]	[1,-,-]	[∞,∞,-]	[∞,∞,-]	[∞,∞,-]	[∞,-,-]	[∞,-,-]
Iteration 2	[2,∞,-]	[2,∞,-]	[∞,∞,∞]	0	1	[2,-,-]	1	[∞,∞,-]	[∞,∞,-]	[∞,∞,-]	[∞,-,-]	[∞,-,-]
Iteration 3	2	2	[∞,3,∞]	0	1	2	1	[∞,3,-]	[3,3,-]	[∞,∞,-]	[3,-,-]	[3,-,-]
Iteration 4	2	2	3	0	1	2	1	3	3	[∞,4,-]	3	3

(a) The value of each vertex during each iteration in machine $\mathcal{M}_1$

Iteration	H	K	N	P	J	M	G	I	L	O	A	F
	CPU/GPU	CPU/GPU	CPU/GPU	CPU/GPU	CPU/GPU	CPU/GPU	CPU	CPU	CPU	CPU	Mirror vertices	
Initialization	[∞,-,-]	[∞,∞,-]	[∞,∞,-]	[∞,-,-]	[∞,-,∞]	[∞,∞,∞]	[∞,-,∞]	[∞,-,∞]	[∞,-,∞]	[∞,∞,∞]	[∞,-,-]	[∞,-,-]
Iteration 1	[∞,-,-]	[∞,∞,-]	[∞,∞,-]	[∞,-,-]	[∞,-,∞]	[∞,∞,∞]	[∞,-,1]	[∞,-,∞]	[∞,-,∞]	[∞,∞,∞]	[∞,-,-]	[∞,-,-]
Iteration 2	[2,-,-]	[2,∞,-]	[∞,∞,-]	[2,-,-]	[∞,-,∞]	[∞,∞,∞]	1	[2,-,∞]	[∞,-,∞]	[∞,∞,∞]	[2,-,-]	[∞,-,-]
Iteration 3	2	2	[∞,3,-]	2	[∞,-,3]	[∞,∞,3]	1	2	[3,-,3]	[∞,3,∞]	2	[∞,-,-]
Iteration 4	2	2	3	2	3	3	1	2	3	3	2	[4,-,-]

(b) The value of each vertex during each iteration in machine $\mathcal{M}_2$

5.2 Optimizations for Cooperative Executor

With the above distributed graph processing procedure, multiple machines (each of them has CPU and GPU processors) in the distributed cluster cooperatively execute the graph algorithm to derive the final results on Nezha. In this section, we propose several performance optimization techniques to improve the efficiency of the cooperative executor by overlapping PCIe transfer, RDMA network communication and CPU/GPU computation efficiently.

Returning to the graph processing example in Section 5.1, there is an active vertex set at the beginning of each iteration. We first categorize these vertices (or the corresponding tasks) into two types: CPU-only and CPU-GPU. We next present the designed optimizations for the processing procedure of each type. We last demonstrate the effectiveness of our techniques by analyzing the processing procedure of the 2nd iteration in $\mathcal{M}_1$.

CPU-only vertices. Most existing studies use the *Scatter-Gather* model to process CPU-only vertices. In particular, studies [31, 43, 50] scatter vertex values and wait until all messages that update the local vertices have been received on each machine. However, this approach fails to maximize resource utilization because communication and computation can be overlapped. To overcome this, Gemini [71] assigns the computation and communication of a task to different threads. However, it wastes computational resources by using dedicated threads for communication across machines. In addition, it cannot fully utilize our proposed RDMA communication adapter in Section 4.3. Thus, we propose a threshold-triggered *Scatter-Gather* model for the CPU-only vertices in Nezha, which efficiently overlaps RDMA communication and graph computation.

Threshold-triggered *Scatter-Gather* model in Nezha: Algorithm 2 shows the detailed design of the threshold-triggered *Scatter-Gather* model in Nezha. Rather than synchronously waiting for all messages to execute the Gather function, we asynchronously probe the messages that have been received, and execute the Gather function with them (see Lines 7 to 9 in Algorithm 2). Specifically, the recvThreshold (see Line 6) can be dynamically adjusted based on the received 'hurried' messages, as detailed in the congestion control method in Section 4.3. It also benefits from the asynchronous design of the RDMA communication protocol in Section 4.3. The major advantages are twofold: (i) it is lightweight as the communication can be done on the local thread and does not trap into the OS kernel, and (ii) it exploits the graph-based RDMA communication adapter and fully utilizes the communication ability of RNIC for distributed graph processing.

Algorithm 2: Scatter-Gather model for CPU-only vertex

```
count ← 0;
foreach vertex v in thread task queue Q do
    foreach neighbor u of v do
        count ++;
        /* Scatter sends the values of updated mirror vertices via RDMA   */
        Scatter(v, u);
        /* Congestion control                                              */
        if count ≥ recvThreshold then
            /* Invoking recv to receive msg                                */
            rdma->recv(&msgs);
            /* Gather updates local vertices by the received msgs         */
            Gather(msgs);
            count ← 0;
/* Gather updates vertices by the received msg                             */
rdma->recv(&msgs);
Gather(msgs);
```

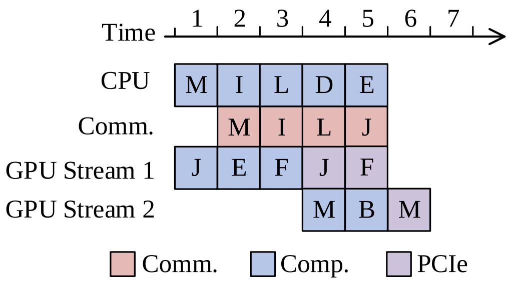


Fig. 7. Breakdown of 2nd iteration in $\mathcal{M}_1$

CPU-GPU vertices. The CPU-GPU vertices can be executed on both CPU and GPU as both CPU memory and GPU memory have the corresponding vertices and edges, see Figure 5. Following the existing study [7], we design a simple-yet-effective GPU invoking strategy to use GPU, i.e., the GPU will be invoked if and only if there are sufficient CPU-GPU tasks in this iteration. Once the GPU is invoked in an iteration, Nezha will initialize the tasks for GPU execution and synchronize the executed results from GPU to CPU via PCIe before sending the message in the Scatter function. Existing CPU-GPU graph processing system [7] invokes GPU to execute vertices and synchronizes the intermediate results in an iteration-based manner. This approach is not efficient as GPU execution and PCIe transfer can be overlapped to improve the graph processing performance. Inspired by Graphreduce [45], which transfers different subgraphs of the raw graph to the GPU during the execution by GPU streaming technique. We devise a streaming synchronization method in Nezha to hide the PCIe transfer costs.

Streaming synchronization method in Nezha: Referring to the 2nd iteration of running BFS on Nezha in Table 3, there are two active vertices D and E in the GPU of $\mathcal{M}_1$, and the values of their neighbors will be computed in the GPU, i.e., the values of vertices J, E, F, M and B. We can assign

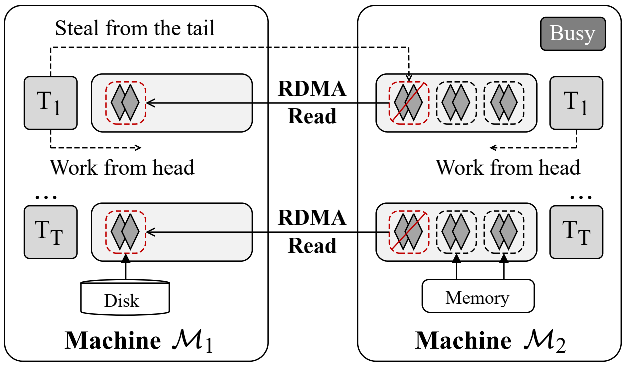


Fig. 8. Inter-machine workload balancing method

the execution of J, E and F to stream 1, and M and B to stream 2. Then, during transferring the updated values of J and F from the GPU to the CPU, the GPU can execute M and B simultaneously.

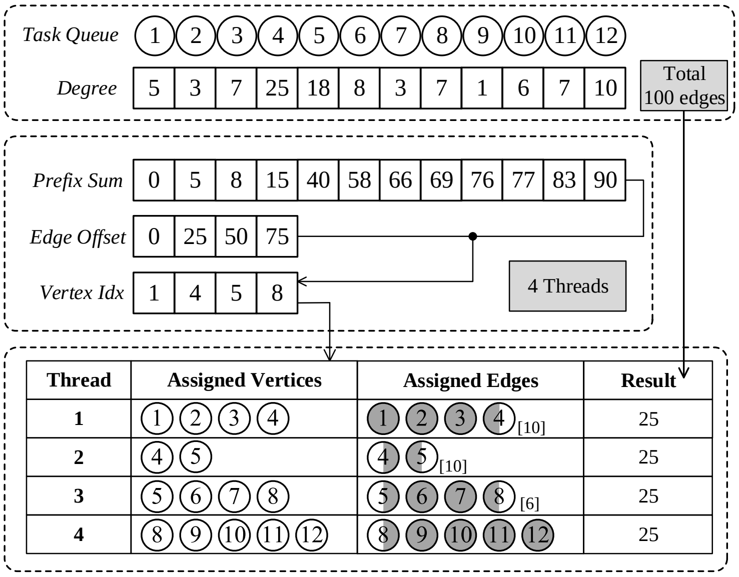
We use the example in Figure 7 to illustrate the overlapped RDMA communication costs, CPU/GPU computation costs and PCIe costs in machine $\mathcal{M}_1$ during the distributed graph processing in Nezha. Specifically, it shows $\mathcal{M}_1$'s 2nd iteration of running BFS on Nezha. For the sake of presentation, the CPU or GPU processes exactly one vertex in each unit of time. RDMA and PCIe transfer a vertex in each unit. The message batch size is 1 and the CPU execution thread sends the message (updated value of the vertex) immediately after execution. Besides, J, E and F are managed by GPU stream 1 while M and B are managed by GPU stream 2. According to Figure 7, we have two observations. Firstly, the RDMA communication costs are overlapped by the CPU computation costs (see the first two rows in Figure 7) in Nezha. Secondly, the PCIe costs are hidden by the streaming synchronization method in $\mathcal{M}_1$ as J and F are transferred to the CPU when M and B are computing on the GPU.

6 Workload Balancer

With the proposed RDMA communication adapter and multi-device cooperative executor in Sections 4 and 5, we can efficiently process the graph using both CPU and GPU in a distributed cluster. However, real-world graphs often exhibit power-law degree distribution [5, 13] and unpredictable data access pattern, which cause serious workload imbalance issues in distributed graph processing systems. In this section, we further improve the performance of Nezha by balancing the workload at inter- and intra-machine levels in Sections 6.1 and 6.2, respectively. In addition, the workload balancer in Nezha focuses on the uneven distribution of the vertex tasks in each iteration during the processing of a specific graph algorithm, e.g., BFS or SSSP with a given root vertex.

6.1 Inter-machine Workload Balancing

We balance the workload among different machines in Nezha as the distribution of workload across machines in a distributed cluster varies significantly. The general idea of our inter-machine workload balance scheme is as follows. In particular, each machine initially processes the workload of the assigned subgraph. Due to the initial workload imbalance issue in Nezha, which is caused by the graph partition approach, the faster machine will take on a portion of the workload from the slower machine to balance the inter-machine workload in the cluster. As depicted in Figure 8, machine $\mathcal{M}_1$ is the faster machine, its working thread T_1 invokes RDMA *READ* to steal a batch of tasks (i.e., vertices) from the busy and slower machine $\mathcal{M}_2$. The atomic operation will be used to mark the stolen tasks to avoid repeated executions on different machines. Since each machine only



Thread	Assigned Vertices	Assigned Edges	Result
1	1 2 3 4	1 2 3 4 [10]	25
2	4 5	4 5 [10]	25
3	5 6 7 8	5 6 7 8 [6]	25
4	8 9 10 11 12	8 9 10 11 12	25

Fig. 9. Vertex distribution strategy

stores a portion of the entire graph in memory, if the stolen tasks cannot find their corresponding edges and vertices in local memory, they will access them from the entire graph on disk.

6.2 Intra-machine Workload Balancing

With the coarse-grained workload balancing at the inter-machine level, we next present our approach to provide a fine-grained workload balancing at the intra-machine level in Nezha. We elaborate on the intra-machine workload balancing scheme by considering both types of tasks, i.e., CPU-only and CPU-GPU.

CPU-only vertices. Inspired by existing static load balancing methods [31] and dynamic load balancing methods [13, 35, 52, 71], the core idea of our balancing scheme is to first assign the vertices among threads and then balances the workload by work stealing during execution. Figure 9 illustrates the vertex assignment method among CPU threads for each iteration. First, it computes the prefix sum of degrees for these CPU-only tasks. Second, it calculates the expected number of processed edges for each thread. For example, each thread in Figure 9 expects to execute 25 edges (which is 100/4). Third, it uniformly assigns these vertices to different working threads based on the prefix sum, which may split the edges of a task across different threads. For example, the first 10 edges of task 4 are assigned to thread 1, while its last 15 edges are assigned to thread 2. In addition, the intra-machine workload balancing scheme adopts the V/E-Steal work stealing method from [7] to balance the workload among CPU cores. In particular, it first determines a stealing order for each thread. When a thread completes its assigned tasks, it will steal vertices or edges from other threads following this order.

CPU-GPU vertices. The computation of CPU-GPU vertices is first assigned to each block of the GPU. The strategy is similar to that of CPU-only vertices in Figure 9. These vertices are maintained on the GPU, and GPU streams are bound to these vertices for computation, as shown in Section 5.2. To balance the workload between CPU and GPU, when all CPU threads finish their computation, they will process the CPU-GPU vertices to balance the workload between CPU and GPU, following the on-demand allocation strategy in [7].

Table 4. Statistics of the graph datasets

Graph	|V|	|E|	D_{avg}	Size
friendster	124.83M	1,806.07M	14.56	15GB
un-union	133.64M	5,955.18M	43.47	50GB
rmat-53	537.27M	20,884.87M	38.91	168GB
clubeweb12	978.41M	42,574.11M	43.53	325GB
rmat-124	1,247.33M	58,658.68M	47.04	450GB

7 Experimental Evaluation

In this section, we evaluate the performance of Nezha by conducting comprehensive experimental studies. We introduce the experimental setting in Section 7.1, present the overall performance evaluation results in Section 7.2, and verify the effectiveness of our proposed techniques in Nezha in Section 7.3.

7.1 Experimental Settings

Graph Dataset. Table 4 summarizes the statistics of 5 graph datasets used during the experimental evaluation, their sizes range from 15 GB to 450 GB. Three of these datasets are real-world datasets: *friendster*, *un-union*, and *clueweb12*. The graphs with the prefix "*rmat*" are generated via PaRMAT [22] to simulate real-world graph characteristics, such as power-law degree distribution. We preprocess all datasets by removing duplicate edges and self-loops, following [7]. In addition, we convert the undirected graphs into directed ones by representing each undirected edge as a pair of directed edges.

Comparison of Distributed Graph Processing Systems. We compared Nezha with the following three existing distributed graph processing systems. We did not compare with Gram [58], a CPU-only distributed graph processing system with RDMA as it is not an open-source project and we cannot get the code from the authors.

- Gemini [71]. It is a CPU-only distributed graph processing system, which is designed for modern multi-core cluster machines.
- Lux [17]. It is a distributed multi-GPU system that achieves efficient graph processing by exploiting locality and the aggregate memory bandwidth of GPUs.
- DRONE [68]. It is a novel CPU-only subgraph-centric distributed graph computing framework.

Evaluated Graph Algorithms. In this work, we test four representative graph algorithms: breadth-first search (BFS), single-source shortest path (SSSP), weakly connected components (WCC), and PageRank (PR). To provide fair comparisons, all systems use the same root vertices with a non-zero out-degree, which are selected randomly from the vertex set. In addition, since SSSP runs on a weighted graph, we randomly assign weights in (0, 1000] to each edge, following the setting of the existing system [7]. For PR, we adopt the settings in Gemini, i.e., utilizing a terminal condition with a damping factor of 0.85 and running it for 20 iterations. The reported measurements are the average results of running the corresponding algorithm 10 times.

Implementation and Hardware Configurations. Nezha is implemented with more than 18,000 lines of C++ and CUDA code. To compile the code, we used GCC version 11.4.0 for C++ code and NVCC version 12.3 for CUDA code. All code is compiled on Ubuntu 18.04 with O3-level optimization.

Table 5. Execution time (in seconds) of compared systems

Alg.	Graph	Gemini	Lux	DRONE	Nezha	Speedup
BFS	*friendster*	4.69	3.12	5.67	**2.45**	$\geq 1.27X$
	un-union	6.54	7.01	11.27	**3.47**	$\geq 1.88X$
	rmat-53	76.08	137.70	92.06	**30.31**	$\geq 2.51X$
	clubeweb12	178.63	—	257.43	**72.39**	$\geq 2.47X$
	rmat-127	473.62	—	—	**224.45**	$\geq 2.11X$
SSSP	*friendster*	23.23	17.47	29.21	**6.64**	$\geq 2.63X$
	un-union	9.86	8.13	18.73	**4.53**	$\geq 1.79X$
	rmat-53	194.53	—	288.154	**46.20**	$\geq 4.21X$
	clubeweb12	478.21	—	—	**121.38**	$\geq 3.94X$
	rmat-127	—	—	—	**286.05**	—
WCC	*friendster*	22.75	14.75	12.13	**10.27**	$\geq 1.18X$
	un-union	20.00	12.45	8.73	**6.02**	$\geq 1.45X$
	rmat-53	204.75	131.57	99.85	**42.67**	$\geq 2.34X$
	clubeweb12	195.10	—	157.63	**68.12**	$\geq 2.31X$
	rmat-127	577.63	—	—	**183.96**	$\geq 3.14X$
PR	*friendster*	92.31	58.91	127.63	**28.51**	$\geq 2.06X$
	un-union	8.47	10.25	21.73	**6.65**	$\geq 1.27X$
	rmat-53	980.47	543.32	1127.35	**188.00**	$\geq 2.89X$
	clubeweb12	1122.92	—	1663.78	**361.06**	$\geq 3.11X$
	rmat-127	3026.24	—	—	**910.51**	$\geq 3.32X$

All experiments are conducted on a cluster which has 6 machines. In particular, each machine includes a 10-core Intel(R) Xeon(R) Silver 4210 CPU, 128 GB of DRAM, an NVIDIA Tesla T4 GPU with 2560 CUDA cores and 16GB of GDDR5 global memory, along with a Mellanox ConnectX-3 56 Gbps InfiniBand NIC connected via PCIe 3.0 x8 to a Mellanox IB Switch.

7.2 Overall Performance Evaluation

Table 5 presents the end-to-end processing time of all evaluated graph systems when running 4 different algorithms on 5 graph datasets. The "Speedup" column shows the speedup factor of Nezha over the fastest existing system in each case. Firstly, there is no doubt that Nezha performs the best among all compared systems. This benefit comes from the efficient RDMA communication adapter introduced in Section 4, the effective multi-device cooperative executor introduced in Section 5, and the effective workload balancer introduced in Section 6. Secondly, Lux encounters runtime errors during graph processing when dealing with large graphs, such as those larger than *rmat-53*, as indicated by the "—" values in Table 5. These errors occur because the data size exceeds the GPU memory capacity. DRONE fails to process the largest dataset *rmat-127* for all algorithms, and *clubeweb12* for SSSP because the consumed memory exceeds the CPU memory capacity during execution. Gemini cannot process the largest dataset *rmat-127* for the SSSP algorithm because its memory usage is at least twice that of the original graph size to support its propagation model. However, our system Nezha returns correct results for all tested graphs, which also confirms the ability of Nezha to process large-scale graphs.

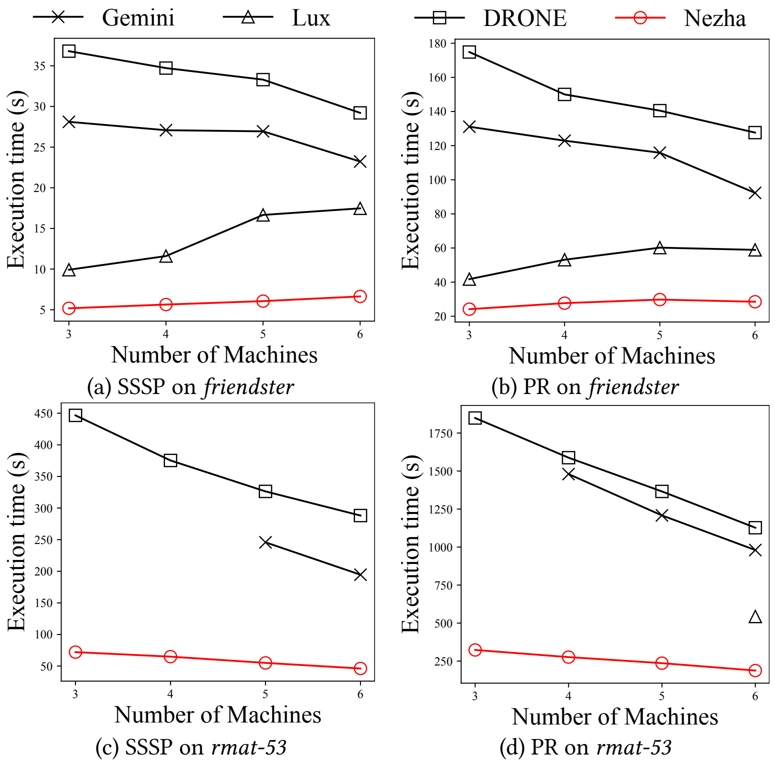


Fig. 10. Execution time w.r.t the number of machines

7.3 Effectiveness Study on Nezha

We verify the effectiveness of the proposed techniques (e.g., communication mechanism, congestion control method) in this section.

7.3.1 Effect of the Number of Machines in a Cluster. We first investigate how the performance of every evaluated system is affected by varying the number of machines. We conduct these experiments by executing the SSSP and PR algorithms on two datasets: a small graph, *friendster*, and a larger graph, *rmat-53*. The tested cluster sizes range from 3 to 6. Figure 10 illustrates the execution time of all compared systems with different number of machines.

Firstly, it is obvious that Nezha performs the best among the compared distributed systems in all cases. Secondly, the performance of all CPU-only distributed systems improves as the number of machines increases for the large graph *rmat-53*. The reason is that the computation power scales as the number of machines increases. However, Lux becomes slower as the number of machines increases because the overhead of initiating an additional GPU outweighs the benefits gained when processing the small graph *friendster*. Nezha mitigates such degradation by efficiently co-operating CPU and GPU, as introduced in Section 5. Thirdly, as shown by the missing data points in Figure 10(c) and Figure 10(d) shown, Lux cannot return the result of the large graph *rmat-53* when the number of machines is insufficient. The reason is GPU memory size is limited and cannot process the large graph in Lux. Nevertheless, Nezha and Lux outperform Gemini and DRONE among the tested cases where all compared systems could return correct results, which confirms that the high bandwidth of GPUs can be used to accelerate the performance of graph processing.

Table 6. Execution time (in seconds) of different RDMA communication methods

Algorithm	Graph	Nezha-Wu	Nezha	Speedup
SSSP	*friendster*	8.43	6.64	1.27X
	uk-union	5.03	4.53	1.11X
	rmat-53	63.76	46.20	1.38X
	clubeweb12	152.93	121.38	1.26X
	rmat-128	—	286.05	—
PR	*friendster*	37.35	28.51	1.31X
	uk-union	7.62	6.65	1.15X
	rmat-53	266.96	188.00	1.42X
	clubeweb12	487.431	361.06	1.35X
	rmat-128	1283.11	910.51	1.41X

Table 7. Execution time (in seconds) w.r.t congestion control

Algorithm	Graph	Nezha-DCC	Nezha	Speedup
SSSP	*friendster*	6.50	6.64	0.98X
	uk-union	4.53	4.53	1.00X
	rmat-53	54.90	46.20	1.19X
	clubeweb12	128.66	121.38	1.06X
	rmat-128	331.82	286.05	1.16X
PR	*friendster*	30.12	28.51	1.06X
	uk-union	6.89	6.65	1.04X
	rmat-53	223.72	188.00	1.19X
	clubeweb12	411.61	361.06	1.14X
	rmat-128	1065.29	910.51	1.17X

7.3.2 Effect of RDMA Communication Mechanism. In this section, we demonstrate the effect of our RDMA communication mechanism by comparing it with that the method proposed in Wukong+G [62], which is an RDMA-based distributed RDF query system. We evaluate the RDMA communication techniques of Wukong+G by replacing the communication component in Nezha with it, and we measure the end-to-end execution time for SSSP and PR across all five graph datasets.

Table 6 shows the evaluated results of two system configurations, where Nezha-Wu integrates Nezha with the RDMA communication method of Wukong+G. Firstly, we observe that Nezha consistently outperforms Nezha-Wu across all settings. The major reason is that the RDMA communication protocol of Nezha is optimized by considering the properties of distributed graph processing (see Section 4.3). Secondly, Nezha-Wu fails to process SSSP on *rmat-128* due to excessive memory consumption (see '—' in Table 6). In particular, Nezha-Wu consumes 16 GB of memory on each machine for RDMA communication, while Nezha consumes only 1GB as Nezha is equipped with a graph-based RDMA communication adapter, which uses less memory for the new connection mode and achieves high memory buffer utilization.

Effect of Congestion Control Method. Table 7 shows the execution time of Nezha with and without the congestion control method, where "Nezha-DCC" indicates Nezha with disabled congestion control method as described in Section 4.3. First of all, Nezha performs better than Nezha-DCC,

Table 8. Execution time (in seconds) w.r.t inter-machine load balancing method

Algorithm	Graph	Nezha-DILB	Nezha	Speedup
SSSP	*friendster*	7.37	6.64	1.11X
	uk-union	4.71	4.53	1.04X
	rmat-53	55.90	46.20	1.21X
	clubeweb12	145.656	121.38	1.20X
	rmat-128	397.61	286.05	1.39X
PR	*friendster*	37.35	28.51	1.03X
	uk-union	6.45	6.65	0.97X
	rmat-53	231.24	188.00	1.23X
	clubeweb12	33.272	361.06	1.20X
	rmat-128	1192.77	910.51	1.31X

Table 9. Execution time and cache miss ratio of intra-machine workload balancing methods

Alg.	Graph	V/E-Steal		Nezha method	
		time	cache miss	time	cache miss
SSSP	*friendster*	10.81	29.11%	6.64	12.37%
	uk-union	6.38	19.87%	4.53	11.25%
	rmat-53	75.77	30.08%	46.20	12.38%
	clubeweb12	189.42	26.38%	121.38	11.73%
	rmat128	486.02	34.28%	291.03	13.89%
PR	*friendster*	44.19	27.11%	28.51	10.66%
	uk-union	8.45	17.87%	6.65	9.11%
	rmat-53	291.41	34.08%	188.00	10.74%
	clubeweb12	483.82	21.38%	361.06	10.73%
	rmat128	1432.18	31.28%	910.51	12.47%

which confirms the effectiveness of the congestion control method in the RDMA communication adapter. In addition, the performance gap between Nezha-DCC and Nezha becomes evident when the processing graph is synthetic, i.e., *rmat-53* or *rmat-128*, as the synthetic graphs exhibit the power-law distribution and incur irregular communication patterns.

7.3.3 Effect of Workoad Balancer. We evaluate the effectiveness of the workload balancer in Nezha in two ways: (i) inter-machine level and (ii) intra-machine level.

Effect of Inter-machine Workload Balancing Method. In this experiment, we compare the overall execution time of Nezha with and without the inter-machine workload balancing method by running SSSP and PR on five datasets. Table 8 shows the execution time and the speedup achieved by the inter-machine workload balancing method, where "Nezha-DILB" is disabling the inter-machine workload balancing method in Nezha. Nezha outperforms Nezha-DILB in all tested cases. This improvement is due to the better utilization of computational resources by stealing tasks from slower machines to faster ones during the execution.

Effect of Intra-machine Workload Balancing Method. We compare our intra-machine workload balancing method in Nezha with the V/E-Steal method in *CGgraph* [7]. In particular, our method in Nezha first uniformly assigns tasks to CPU threads, then employs the V/E-Steal method to adjust the workload dynamically. We measure the execution time and cache miss ratio of our

Table 10. Execution time (in seconds) by varying batch size

Alg.	Graph	16	32	64	128	256	512
SSSP	*friendster*	7.0	6.8	6.9	**6.6**	6.9	7.1
	uk-union	4.3	4.3	**4.2**	4.53	5.2	5.6
	rmat-53	60.2	53.5	50.1	**46.2**	57.2	57.4
	clubeweb12	131.2	124.8	125.5	**121.3**	129.6	135.4
	rmat-128	315.8	311.5	291.0	286.0	**279.5**	298.7
PR	*friendster*	32.17	31.22	29.15	28.51	**27.89**	28.24
	uk-union	6.9	6.9	7.0	**6.7**	6.8	7.0
	rmat-53	195.4	189.5	193.1	**188.0**	196.7	197.2
	clubeweb12	374.5	371.6	367.4	361.1	**354.3**	381.3
	rmat-128	930.2	923.5	917.5	**910.5**	913.3	927.6

proposed method and the V/E-Steal method by running SSSP and PR on five datasets. Obviously, our workload balancing method outperforms V/E-Steal on both cache ratio and execution time, as shown in Table 9. The core reason is that uniformly assigning the edges at the beginning of each iteration reduces the overhead for each thread to steal tasks from overloaded threads.

7.3.4 Effect of Design Choices In this section, we evaluate the effect of different design choices (i.e., message batch size, "hurried" threshold) on the end-to-end performance of Nezha.

Message Batch Size. Table 10 illustrates the execution time of the SSSP and PR algorithms on five graph datasets. The message batch size varies from 16 to 512. We observe that as the batch size increases, the execution time decreases first and then increases later. The reason is that the overhead of sending many small messages is too large when the batch size is small, while it wastes bandwidth when the batch size is large as a message will not be sent until the batch is full. Hence, we set the batch size of Nezha to 128 by default, as it performs best in most test cases.

The "hurried" Threshold. Referring to Section 4.3, when the messages between the *Clear* pointer and the *Send Tail* pointer occupy too large space, the subsequent messages will be marked as 'hurried'. To illustrate its effect and to determine the appropriate threshold, we run the SSSP and PR algorithms on five graph datasets by varying the hurried threshold from 50% to 90% of the send buffer capacity. The experimental results are shown in Table 11. Obviously, too small or too large hurried threshold value is not optimal as a small threshold leads to frequent updates of the *Clear* pointer via RDMA communication, while a large threshold causes delayed notification, resulting in the sender is being blocked. Hence, we set the hurried threshold to 80% by default.

In addition, the default settings of message buffer size and the "hurried" threshold in Nezha can be determined by the users based on internal testing for their specific graph applications.

8 Related Work

In this section, we review the most relevant studies related to Nezha from three aspects: graph processing systems, RDMA-based systems, and communication optimization techniques.

Graph Processing Systems. Table 12 provides a brief summary of representative existing graph processing systems. In particular, we categorize them based on whether they are distributed systems, the processors they use, and the communication networks they employ.

Single-machine CPU-based: Single-machine CPU-based graph processing systems [34, 47, 67] optimize the CPU utilization for processing graphs that can fit into the CPU main memory. Ligra [47] presents a lightweight framework that employs a dual update propagation model. Galois [34] follows

Table 11. Execution time (in seconds) by varying 'hurried' threshold

Alg.	Graph	50%	60%	70%	80%	90%
SSSP	*friendster*	6.84	6.81	6.83	**6.64**	6.69
	uk-union	4.53	4.52	**4.51**	4.53	4.53
	rmat-53	55.24	50.29	48.37	**46.20**	46.21
	clubeweb12	128.56	122.88	122.74	**121.38**	122.00
	rmat128	312.45	310.86	297.66	**286.05**	291.01
PR	*friendster*	29.13	**28.01**	28.50	28.51	29.13
	uk-union	**6.11**	6.64	6.64	6.65	6.72
	rmat-53	220.42	204.53	195.33	**188.00**	192.45
	clubeweb12	382.59	380.55	367.67	**361.06**	374.25
	rmat128	993.01	935.78	921.51	**910.51**	933.27

an asynchronous graph processing mode and enables lock-free computation, leading to efficient parallelism and reduced synchronization overhead.

Single-machine GPU-based: With the advancement of General-Purpose Computing on Graphics Processing Units (GPGPU), GPUs are increasingly leveraged for high-performance graph processing due to their data-parallel capabilities. Single-GPU graph processing systems [23, 57] utilize the GPU to process graph data efficiently. Cusha [23] optimizes the graph representation for efficient GPU resource utilization. Gunrock [57] introduces a frontier-based model to balance the workload among GPU threads. Single-node multi-GPU systems [3, 37, 70] leverage multiple GPUs in a single node to cooperatively process graph data. Different GPUs communicate via PCIe in these systems. For example, Groute [3] implements a GPU-based asynchronous execution and communication approach in an 8-GPU machine.

Single-machine CPU-GPU-based: In recent years, several graph systems utilize both CPU and GPU for efficient graph processing. For example, LargeGraph [69] partitions graph data into frequent paths for GPU execution and infrequent paths for CPU execution. CGgraph [7] implements an on-demand task allocation mechanism to ensure balanced workload distribution between the CPU and GPU. These systems confirm that well-designed CPU-GPU collaborative processing can significantly improve performance by fully utilizing the computational power of both processors.

Distributed-cluster CPU-based: With the growth of the graph size, several distributed graph systems have been proposed in both academia and industry. Gemini [71] introduces a chunk-based partitioning strategy that enables low-overhead scaling and employs a fine-grained work-stealing approach to maintain workload balance. DRONE [68] adopts a subgraph-centric model for processing in distributed clusters. GraM [58] utilizes a message-passing mechanism, derived from FaRM [9], designed for efficient transaction processing with RDMA. Lux [17] accelerates graph processing by taking advantage of memory hierarchy locality in multi-GPU clusters.

In this work, we propose Nezha, which differs from existing system as it is the first system that leverages both CPU and GPU for graph processing in a distributed cluster (see Table 12).

RDMA-based Systems. With the rapid development of Remote Direct Memory Access (RDMA), many systems employ RDMA to accelerate the communication costs [18] in various domains. However, RDMA is a lower-level interface that requires workload-specific design to achieve good performance [20, 72]. Thus, many systems propose their designs to utilize RDMA properly. For example, FaRM [9] introduces an RDMA communication stack for efficient transaction processing, which utilizes versioned objects and locks to ensure data consistency. GraM [9] employs the

Table 12. Existing representative graph processing systems

System	Platform	Processor	Network
Ligra [46]	Single	CPU	-
Galois [34]	Single	CPU	-
Cusha [23]	Single	GPU	-
Groute [3]	Single	GPU	-
Largegraph [69]	Single	CPU-GPU	-
CGgraph [7]	Single	CPU-GPU	-
Powergraph [13]	Distributed	CPU	TCP/IP
Gemini [71]	Distributed	CPU	TCP/IP
DRONE [68]	Distributed	CPU	TCP/IP
GraM [58]	Distributed	CPU	RDMA
Lux [17]	Distributed	GPU	TCP/IP
Our **Nezha**	**Distributed**	**CPU-GPU**	**RDMA**

message-passing mechanism of FaRM to enhance efficiency and scalability in distributed graph processing. Wukong+G [62] uses RDMA to accelerate its Resource Description Framework(RDF). Moreover, RDMA has been used in various applications in distributed environment, e.g., distributed joins [2], RDMA-based shuffle operations [27], and indexes implementations [73].

In this work, Nezha utilizes RDMA for distributed graph processing. However, the proposed RDMA techniques in the aforementioned works cannot be directly adapted in Nezha to achieve good performance, as they do not consider the characteristics of distributed graph processing. On the contrary, we propose a graph-based RDMA communication adapter for Nezha in this work.

Communication Optimizations. The communication costs of distributed systems can be improved by optimizing either communication messages [31, 40, 44] or communication mechanisms [71]. However, these proposed techniques are not sufficient for Nezha.

9 Conclusion

In this work, we propose Nezha, an efficient distributed graph processing system on heterogeneous hardware (i.e., CPU, GPU, and RDMA). We first propose an RDMA communication adapter for efficient communication during distributed graph processing. Then, we introduce a multi-device cooperative executor, which enables graph processing across multiple machines, each equipped with the CPU and GPU. This executor efficiently overlaps computation time, PCIe transfers, and network communication. Next, we propose a workload balancer to further enhance overall performance by balancing the workload both within and across machines. We conduct extensive experiments with Nezha to demonstrate its superiority over existing systems. There are two promising directions for future work: (i) extending Nezha to process more complex graph analysis applications (e.g., pattern matching); and (ii) adapting the proposed techniques (e.g., RDMA connection mode/communication protocol) in Nezha to other distributed systems.

Acknowledgments

We are grateful to the anonymous reviewers and the shepherd for their insightful suggestions. Dr. Ye Yuan is supported by the National Key RD Program of China(Grant No.2022YFB2702100) and the NSFC (Grant Nos. 61932004, 62225203, U21A20516). Dr. Bo Tang and Haotian Liu were supported by National Science Foundation of China (NSFC No. 62422206) and a research gift from AlayaDB Inc. Pengjie Cui and Haotian Liu contributed equally to this work.

References

[1] Andreea Anghel, German Rodriguez, Bogdan Prisacari, Cyriel Minkenberg, and Gero Dittmann. 2015. Quantifying communication in graph analytics. In *HPC*. 472–487.

[2] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-scale in-memory join processing using RDMA. In *SIGMOD*. 1463–1475.

[3] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An asynchronous multi-GPU programming model for irregular computations. *ACM SIGPLAN Notices* 52, 8 (2017), 235–248.

[4] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan RK Ports. 2021. PRISM: Rethinking the RDMA interface for distributed systems. In *SOSP*. 228–242.

[5] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *TOPC* 5, 3 (2019), 1–39.

[6] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *PVLDB* 8, 12 (2015), 1804–1815.

[7] Pengjie Cui, Haotian Liu, Bo Tang, and Ye Yuan. 2024. CGgraph: An Ultra-fast Graph Processing System on Modern Commodity CPU-GPU Co-processor. *PVLDB* 17, 6 (2024), 1405–1417.

[8] Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, Sriram Sankar, et al. 2013. Unicorn: A system for searching the social graph. *PVLDB* 6, 11 (2013), 1150–1161.

[9] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *NSDI 14*. 401–414.

[10] Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, Xiaojian Luo, Qiang Yin, Ping Lu, Yang Cao, and Ruiqi Xu. 2018. Parallelizing sequential graph computations. *TODS* 43, 4 (2018), 1–39.

[11] Philip Werner Frey and Gustavo Alonso. 2009. Minimizing the hidden cost of RDMA. In *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, 553–560.

[12] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. 2012. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *PACT*. 345–354.

[13] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel computation on natural graphs. In *OSDI*. 17–30.

[14] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*. 599–613.

[15] Masatoshi Hanai, Toyotaro Suzumura, Wen Jun Tan, Elvis Liu, Georgios Theodoropoulos, and Wentong Cai. 2019. Distributed edge partitioning for trillion-edge graphs. *arXiv preprint arXiv:1908.05855* (2019).

[16] Hideya Iwasaki, Kento Emoto, Akimasa Morihata, Kiminori Matsuzaki, and Zhenjiang Hu. 2022. Fregel: a functional domain-specific language for vertex-centric large-scale graph processing. *Journal of Functional Programming* 32 (2022), e4.

[17] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A distributed multi-gpu system for fast graph processing. *PVLDB* 11, 3 (2017), 297–310.

[18] Hyun-Wook Jin, Sundeep Narravula, Gregory Brown, Karthikeyan Vaidyanathan, Pavan Balaji, and Dhabaleswar K Panda. 2005. Performance evaluation of rdma over ip: A case study with the ammasso gigabit ethernet nic. In *Workshop on HPI-DC; In conjunction with HPDC-14*.

[19] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. 2017. High-level programming abstractions for distributed graph processing. *TKDE* 30, 2 (2017), 305–324.

[20] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design guidelines for high performance RDMA systems. In *ATC*. 437–450.

[21] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: a system for dynamic load balancing in large-scale graph processing. In *EuroSys*. 169–182.

[22] Farzad Khorasani, Keval Vora, and Rajiv Gupta. 2015. Parmat: A parallel generator for large r-mat graphs. In *PACT*.

[23] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *HPDC*. 239–252.

[24] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi:Large-Scale graph computation on just a PC. In *OSDI*. 31–46.

[25] Hu Li, Tianjia Chen, and Wei Xu. 2016. Improving spark performance with zero-copy buffer management and RDMA. In *INFOCOM WKSHPS*. IEEE, 33–38.

[26] Mingfan Li, Ke Wen, Han Lin, Xu Jin, Zheng Wu, Hong An, and Mengxian Chi. 2019. Improving the performance of distributed mxnet with rdma. *International Journal of Parallel Programming* 47 (2019), 467–480.

[27] Feilong Liu, Lingyan Yin, and Spyros Blanas. 2019. Design and evaluation of an rdma-aware data shuffling operator for parallel database systems. *TODS* 44, 4 (2019), 1–45.

[28] Jiuxing Liu, Jiesheng Wu, Sushmitha P Kini, Pete Wyckoff, and Dhabaleswar K Panda. 2003. High performance RDMA-based MPI implementation over InfiniBand. In *Proceedings of the 17th annual international conference on Supercomputing*. 295–304.

[29] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB* 8, 3 (2014), 281–292.

[30] Teng Ma, Tao Ma, Zhuo Song, Jingxuan Li, Huaixin Chang, Kang Chen, Hai Jiang, and Yongwei Wu. 2019. X-RDMA: Effective RDMA middleware in large-scale production environments. In *CLUSTER*. 1–12.

[31] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD*. 135–146.

[32] Christian Mayer, Ruben Mayer, Muhammad Adnan Tariq, Heiko Geppert, Larissa Laich, Lukas Rieger, and Kurt Rothermel. 2018. Adwise: Adaptive window-based streaming edge partitioning for high-speed graph processing. In *ICDCS*. 685–695.

[33] Ruben Mayer, Kamil Orujzade, and Hans-Arno Jacobsen. 2022. Out-of-core edge partitioning at linear run-time. In *ICDE*. 2629–2642.

[34] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *SOSP*. 456–471.

[35] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. 2012. OpenMP task scheduling strategies for multicore NUMA systems. *IJHPCA* 26, 2 (2012), 110–124.

[36] Anil Pacaci and M Tamer Özsu. 2019. Experimental analysis of streaming algorithms for graph partitioning. In *SIGMOD*. 1375–1392.

[37] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D Owens. 2017. Multi-GPU graph analytics. In *IPDPS*. 479–490.

[38] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. 2015. Hdrf: Stream-based partitioning for power-law graphs. In *CIKM*. 243–252.

[39] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *PVLDB* 11, 12 (2018), 1876–1888.

[40] Louise Quick, Paul Wilkinson, and David Hardcastle. 2012. Using pregel-like large scale graph processing frameworks for social network analysis. In *ASONAM*. 457–463.

[41] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP*. 472–488.

[42] Semih Salihoglu and Jennifer Widom. 2013. Gps: A graph processing system. In *SSDBM*. 1–12.

[43] Semih Salihoglu and Jennifer Widom. 2014. Optimizing graph algorithms on pregel-like systems. (2014).

[44] Nadathur Satish, Narayanan Sundaram, Md Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. 2014. Navigating the maze of graph analytics frameworks using massive graph datasets. In *SIGMOD*. 979–990.

[45] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. 2015. Graphreduce: processing large-scale graphs on accelerator-based systems. In *SC*. 1–12.

[46] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*. 135–146.

[47] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*. 135–146.

[48] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. 2014. Goffish: A sub-graph centric framework for large-scale graph analytics. In *Euro-Par 2014 Parallel Processing*. 451–462.

[49] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *SIGKDD*. 1222–1230.

[50] Philip Stutz, Abraham Bernstein, and William Cohen. 2010. Signal/collect: graph algorithms for the (semantic) web. In *The Semantic Web–ISWC*. 764–780.

[51] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. 2017. Rfp: When rpc is faster than server-bypass with rdma. In *EuroSys*. 1–15.

[52] Adrian Tate, Amir Kamil, Anshu Dubey, Armin Groblinger, Brad Chamberlain, Brice Goglin, Harold C Edwards, Chris J Newburn, David Padua, Didem Unat, et al. 2014. *Programming abstractions for data locality*. Technical Report. OSTI.

[53] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From" think like a vertex" to" think like a graph". *PVLDB* 7, 3 (2013), 193–204.

[54] Shin-Yeh Tsai and Yiying Zhang. 2017. LITE kernel RDMA support for datacenter applications. In *SOSP*. 306–324.

[55] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. Fennel: Streaming graph partitioning for massive scale graphs. In *WSDM*. 333–342.

[56] Xubo Wang, Dong Wen, Lu Qin, Lijun Chang, and Wenjie Zhang. 2022. Scaleg: A distributed disk-based system for vertex-centric graph processing. In *ICDE*. 1511–1512.

[57] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *PPoPP*. 1–12.

[58] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. 2015. Gram: Scaling graph computation to the trillions. In *SoCC*. 408–421.

[59] Mengbai Xiao, Hao Wang, Liang Geng, Rubao Lee, and Xiaodong Zhang. 2019. Catfish: Adaptive RDMA-enabled R-Tree for low latency and high throughput. In *IEEE 39th International Conference on Distributed Computing Systems*. 164–175.

[60] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. Sync or async: Time to fuse for distributed graph-parallel computation. *ACM SIGPLAN Notices* 50, 8 (2015), 194–204.

[61] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB* 7, 14 (2014), 1981–1992.

[62] Zihang Yao, Rong Chen, Binyu Zang, and Haibo Chen. 2021. Wukong+ G: Fast and concurrent RDF query processing using RDMA-assisted GPU graph exploration. *TPDS* 33, 7 (2021), 1619–1635.

[63] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. 2005. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *SC*. 25–25.

[64] Ye Yuan, Guoren Wang, Lei Chen, and Haixun Wang. 2013. Efficient keyword search on uncertain graph data. *TKDE* 25, 12 (2013), 2767–2779.

[65] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A Fault-Tolerant abstraction for In-Memory cluster computing. In *NSDI 12*. 15–28.

[66] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. 2016. The end of a myth: Distributed transactions can scale. *arXiv preprint arXiv:1607.00655* (2016).

[67] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware graph-structured analytics. In *PPoPP*. 183–193.

[68] Shuai Zhang, Zite Jiang, Xingzhong Hou, Mingyu Li, Mengting Yuan, and Haihang You. 2022. DRONE: An Efficient Distributed Subgraph-Centric Framework for Processing Large-Scale Power-law Graphs. *TPDS* 34, 2 (2022), 463–474.

[69] Yu Zhang, Da Peng, Xiaofei Liao, Hai Jin, Haikun Liu, Lin Gu, and Bingsheng He. 2021. LargeGraph: An efficient dependency-aware GPU-accelerated large-scale graph processing. *TACO* 18, 4 (2021), 1–24.

[70] Jianlong Zhong and Bingsheng He. 2013. Medusa: Simplified graph processing on GPUs. *TPDS* 25, 6 (2013), 1543–1552.

[71] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric distributed graph processing system. In *OSDI*. 301–316.

[72] Tobias Ziegler, Viktor Leis, and Carsten Binnig. 2020. RDMA Communciation Patterns: A Systematic Evaluation. *Datenbank-Spektrum* 20 (2020), 199–210.

[73] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing distributed tree-based index structures for fast rdma-capable networks. In *SIGMOD*. 741–758.

Received July 2024; revised September 2024; accepted November 2024