Trilobita: A Distributed Graph Processing System

50.041 Course Project Final Report

Instructor

Sudipta Chattopadhyay

Group 4

Guo Yuchen

Guo Ziniu

Liang Junyi

Wang Yanbao

Xiang Siqi

TABLE OF CONTENT

1 INTRODUCTION....1
2 SYSTEM ARCHITECTURE....2
2.1 Overall Architecture....2
2.2 Major Components....3
2.2.1 Master Server....3
2.2.2 Master Server Replica....3
2.2.3 Worker Server....4
2.2.4 Trilobita Environment....4
3 DESIGN....4
3.1 Features....4
3.1.1 Distributed Graph Processing....5
3.1.2 Fault Tolerance for Worker and Master....5
3.1.3 Functionable Instances....5
3.1.4 Server Performance Monitor....6
3.1.5 Scalable Cluster and Parallelism....6
3.1.6 User-definable Vertex and Task....6
3.1.7 User-definable Functionable Instance....6
3.1.8 User-definable Graph Partition Strategy....7
3.2 Correctness....7
3.2.1 Pregel-like System Correctness Guarantee....7
3.2.2 Implementation of Correctness in Trilobita....7
3.3 Scalability....8
3.3.1 Cluster Size Scalability....8
3.3.2 Single Machine Parallelism Scalability....8
3.3.3 Graph Partition Scalability....8
3.4 Fault Tolerance....9
3.4.1 Definition and Types of Faults....9
3.4.2 Fault Resilience....9
3.4.3 Implementations....10
3.5 Limitations....11
4 EVALUATION....11
4.1 Correctness Evaluation....12
4.1.1 No fault....12
4.1.1 Master Failure....12
4.1.1 Worker Failure....12
4.2 Scalability Evaluation....13
4.3 Performance Evaluation....13
4.3.1 Computation Complexity....14
4.3.2 Graph Scale....15
4.3.3 Evaluation of Different Graph Types....15
5 CONCLUSION....16
6 ADDITIONAL INFORMATION....16

1 INTRODUCTION

Many practical computing problems concern large graphs, where relationships and connections among vast amounts of vertices play an important role in computation. These computing problems span from social network analysis and recommendation systems to neural network inferences. As the scale and complexity of these graphs continue to grow, the need for efficient and scalable graph processing systems becomes significant. To tackle these problems, Google introduced Pregel in 2006, presenting a vertex-centric computation model specifically crafted for distributed graph processing. Despite the detailed explanation provided in the paper, Pregel remains exclusive to Google.

Motivated by the necessity to address these challenges embedded in distributed graph processing and to study distributed systems and gain hands-on experience, we developed *Trilobita* – a Pregel-like distributed graph processing system implemented to handle large-scale graph processing tasks in a distributed environment efficiently.

Throughout the development of *Trilobita*, we adhere to the following guiding principles.

1. *Fault Tolerance*: the system should be able to gracefully manage failures of both the master and worker servers, ensuring continuity in task execution and failure handling.
2. *Flexibility*: users of *Trilobita* should be empowered to customize their graph processing tasks and assemble operational clusters by using their personal laptops.
3. *Consistency*: the system should guarantee causal consistency during the computation and fault-handling process, providing a reliable and predictable environment for users.
4. *Efficiency*: the connection and communication between different machines should be efficient and adapt to varying cluster sizes for optimal performance.

The rest of the report is organized as follows. In Section 2, we present an introduction to the *Trilobita* architecture. In Section 3 we discuss the design of the system, including implemented features, correctness, scalability, fault tolerance, and limitations. Section 4 evaluates the system's performance under varying conditions. Finally, we conclude our report and discuss future improvements.

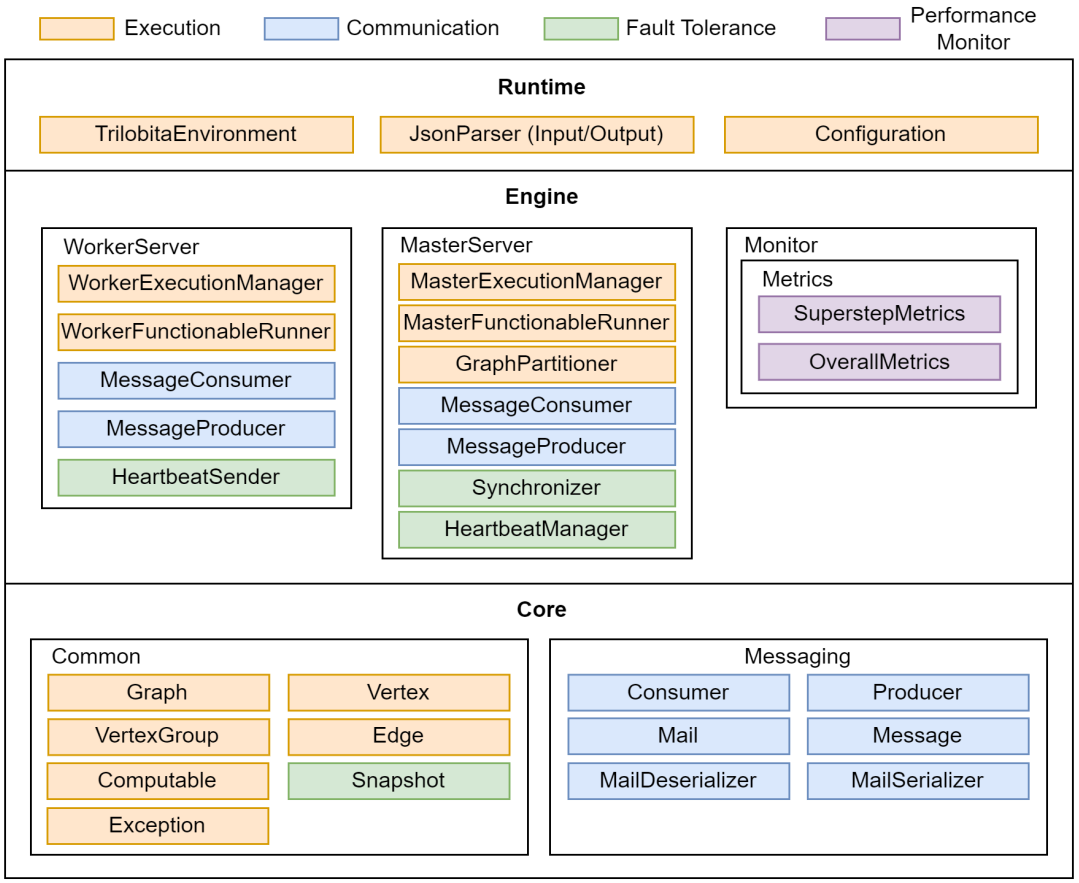
2 SYSTEM ARCHITECTURE

In this section, we discuss the overall architecture of *Trilobita*, followed by a breakdown of major components, providing an overview of their modules and respective functionalities.

2.1 Overall Architecture

The *Trilobita* distributed graph processing system is architectured in three layers, with each layer building upon the functionalities of the preceding one. The overall architecture is illustrated in Figure 1.

1. *Core*. The *Core* layer serves as the foundation, defining essential entities such as Graph, Vertex, Edge, and Snapshot. Additionally, it provides server-to-server messaging APIs to upper layers, facilitating communication between servers.
2. *Engine*. Expanding on the *Core*, the *Engine* layer implements both worker and master servers. It also integrates a side monitor to record the performance of the servers.
3. *Runtime*. The final layer, *Runtime*, defines the functional environment of *Trilobita*. This includes input parsing and configuring computation tasks. Additionally, users will utilize APIs provided by this layer to initiate and start the master and worker servers.

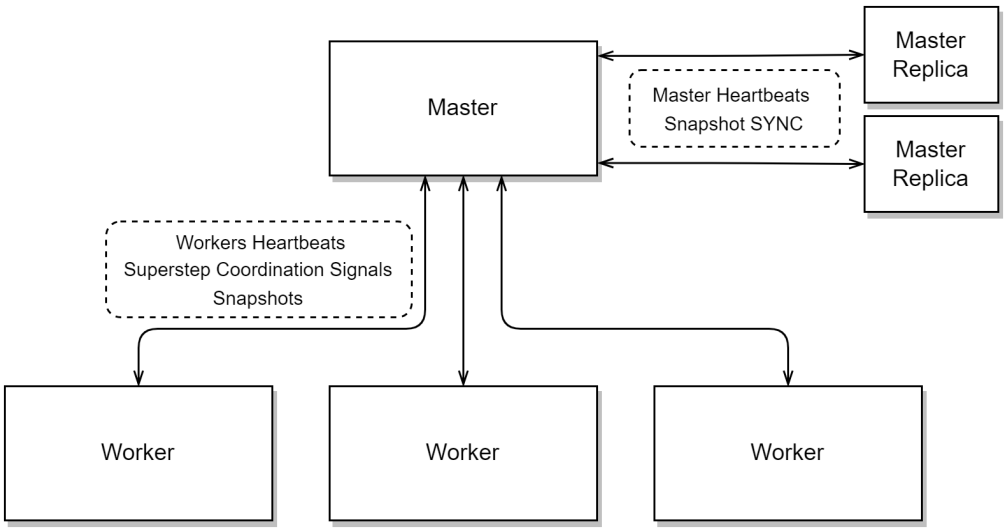


[Figure 1]: System architecture

This structured architecture ensures a systematic progression of functionalities, contributing to the overall robustness, scalability, and efficiency of the *Trilobita* system.

2.2 Major Components

In this section, we provide a comprehensive breakdown of *Trilobita*'s architecture, exploring its major components. Each of these components plays a vital role in the functioning of the system, contributing to its computation correctness, fault tolerance, and scalability. Figure 2 illustrates the overall relationships of the master, workers, and master replicas.



[Figure 2]: Relationships of the master, workers, and master replicas

2.2.1 Master Server

The Master Server is the coordinator of the graph computation process. It is responsible for coordinating superstep executions, broadcasting aggregation messages, periodically synchronizing snapshots with master server replicas, and efficiently handling faults arising from worker server failures.

2.2.2 Master Server Replica

Designed as fail-safe guardians, Master Server Replicas function as backup masters to preserve the system's stability in the event of a master failure. While not actively participating in the actual computation process, these replicas periodically synchronize system snapshots with the master server, monitoring for potential master failures. Upon detecting a master failure, one of the Master Server Replicas takes over and assumes control of the execution, ensuring the system's ongoing continuity and averting disruptions to the computation process.

2.2.3 Worker Server

Worker Servers are responsible for actual computation task executions as well as vertex-level communications. Upon completing tasks in a superstep, Worker Servers transmit completion signals, along with their current states if a snapshot is requested, to the master. They then await the start signal for the subsequent superstep.

2.2.4 Trilobita Environment

To enhance user-friendliness and leverage the system's usability and adaptivity, *Trilobita* introduces the Trilobita Environment component. It serves as an integrated environment that consolidates and simplifies the initiation and launch processes of workers and the master server. This strategic addition empowers users through a set of convenient APIs, offering a friendly approach to configure jobs and initiate servers, thereby enhancing the overall usability of the system.

3 DESIGN

In this section, we discuss the design aspects of *Trilobita*, examining its features, correctness, scalability, fault tolerance, and limitations. The design of any distributed system is a critical determinant of its effectiveness and reliability. *Trilobita*, our distributed graph processing system, has been carefully implemented to meet the demands of distributed computing in a faulty environment, and this section provides a detailed exploration of its key design elements. From the introduction of *Trilobita*'s features to a demonstration of its correctness, we discuss the system's scalability in facing distributed computation environments and its robust fault tolerance mechanisms. Additionally, we discuss the identified limitations of *Trilobita*, providing a comprehensive view of its design considerations.

3.1 Features

We implement a rich set of features in *Trilobita* to empower users with efficiency, reliability, and scalability in distributed graph processing. This section discusses the key features that define the system's capabilities.

Overall, from 3.1.1 to 3.1.4 we discuss the internal features that are implemented in the system. These features, ranging from executions to fault tolerance, showcase *Trilobita*'s prowess in handling computational tasks in a distributed environment. Subsequently, from 3.1.5 to 3.1.8, we introduce user-definable features that are empowered by our carefully designed APIs. These customizable features, including user-definable vertices, tasks,

functionable instances, and graph partition strategies, reflect our commitment to providing users with a flexible and scalable framework for distributed graph processing.

3.1.1 Distributed Graph Processing

Trilobita is designed to efficiently handle graph processing tasks. At task initiation, the master server partitions the graph into vertex groups, assigning each group to a worker. Subsequently, the master coordinates the execution process, controlling the flow of supersteps, and concluding the task once all vertices reach an inactivated state. This streamlined workflow guarantees *Trilobita*'s efficiency and correctness in processing the graph. Notably, *Trilobita* does not impose any constraints on the types of tasks it can handle. As long as the task and vertex classes adhere to the abstract classes provided, *Trilobita* can efficiently and accurately execute a diverse range of tasks.

3.1.2 Fault Tolerance for Worker and Master

Trilobita is implemented to gracefully handle fault tolerance for both workers and the master. This is done by its heartbeat and snapshot mechanism. The detailed fault tolerance implementations are discussed in Section 3.4.

3.1.3 Functionable Instances

Trilobita incorporates functionable instances to manage global information and optimize communication between vertices in a distributed environment. The term 'Functionable' denotes the instances' capability of facilitating the execution and messaging processes in the worker and master servers. Following *Pregel*, *Trilobita* implements two key functionable instances within the system: *Combiner* and *Aggregator*.

1. *Combiner*: A Combiner is a mechanism that optimizes communication among vertices by combining messages before sending them over the network. Messages that are sent to the same vertex are combined into a single message based on user-defined rules, such as sum, min, or max. This functionable component is applied to reduce the volume of data transmitted between vertices, and therefore improves efficiency and eases the messaging overhead.

2. *Aggregator*: An aggregator is a mechanism for aggregating values across all vertices in a distributed graph, which can be used to facilitate global communication and coordination. At the end of a superstep, each vertex contributes its local information to the aggregator, and the aggregator aggregates these contributions to produce a global result based on

user-defined rules. The result will then be made available to all aggregators in the next superstep. This mechanism allows global information to be computed efficiently and shared among all vertices in the system.

3.1.4 Server Performance Monitor

To monitor the processing time and evaluate the performance of the system, we implemented a Monitor. When a task is executed, the Monitor records the execution, messaging, total time, and the number of messages of a superstep in each server. This performance data is systematically stored in the 'Trilobita/data/performance' directory, providing users with valuable insights into the system's processing efficiency and performance.

3.1.5 Scalable Cluster and Parallelism

In *Trilobita*, the scalability of the cluster and the degree of parallelism within each worker server are user-defined and adjustable. Users can modify the cluster size and parallelism level by changing the values of *numOfWorker* and *parallelism* in *TrilobitaEnvironment Configuration*. This provides a flexible and scalable execution environment, allowing users to tailor to the specific requirements of their graph processing tasks.

3.1.6 User-definable Vertex and Task

For customizable graph processing tasks, *Trilobita* provides a set of APIs that empower users to define their vertex and tasks.

1. *Vertex:* Users can define vertices by adhering to the abstract Vertex class provided in the system. This allows users to define the vertex that adapts to different tasks.
2. *Computable*: Considering the existence of non-numerical tasks, such as string concatenations, we implement the Computable interface. This interface allows users to define operations tailored to computation values, maintaining an adaptable framework for diverse graph processing requirements.
3. *GraphParser*: We implemented GraphParser to enable users to parse a graph from a JSON file. This graph can be loaded to *TrilobitaEnvironment* by the *loadGraph* API, and therefore be processed in the *Trilobita* system.

3.1.7 User-definable Functionable Instance

The Functionable interface is designed to be easily customizable, allowing users to implement self-defined logic based on the specific requirements of their graph algorithms. Specifically, the users may choose to define a new Functionable object or to extend the Combiner or Aggregator class by implementing the combine or aggregate functions.

3.1.8 User-definable Graph Partition Strategy

The graph partitioning process is handled by the Partitioner component in the master, which divides the graph based on a chosen partition strategy. These strategies, essential for defining the partitioning logic, are represented as interfaces with a key function, *getServerIdByVertexId*. This function is crucial not only during the partitioning phase but also helps worker servers locate the right server for a specific vertex. We implemented two main partition strategies in the system: hash partition strategy and index partition strategy. Additionally, users have the flexibility to customize their own partition strategy by overriding the *getServerIdByVertexId* function. The new partition strategy can be passed to the graph partitioner through APIs provided in *TrilobitaEnvironment*.

3.2 Correctness

In this section, we discuss the robust correctness guarantee provided by *Trilobita*.

3.2.1 Pregel-like System Correctness Guarantee

A Pregel-like system ensures correctness by adhering to fundamental principles:

- *Superstep Synchronization*: Workers collectively wait until all vertices complete their computations before advancing to the next superstep. This synchronization between supersteps maintains a consistent state across all vertices.
- *Vertex-Centric Programming Model*: Each vertex's computation relies on its local state and incoming messages, isolating the logic of each vertex for independent reasoning.

3.2.2 Implementation of Correctness in Trilobita

Trilobita aligns with the Pregel system, ensuring correctness through key characteristics:

- *Superstep Synchronization*: *Trilobita*, like any Pregel-like system, employs a superstep synchronization mechanism for coordinated execution and data consistency.
- *FIFO Message Passing Rule*: All message passings in one superstep follow the FIFO rule.
- *Vertex-Centric Programming Model*: The computation of each vertex is performed based on its local state and incoming messages.
- *Fault Handling*: Robust fault handling mechanisms are implemented to ensure system resilience and data consistency. A detailed explanation of *Trilobita*'s fault tolerance, including its definition and implementation, will be provided in Section 3.4.

We extend our assessment and detailed evaluation of the correctness guarantee secured by *Trilobita* in Section 4.1. In summary, the design of *Trilobita* guarantees correctness through

the meticulous implementation of Pregel-like correctness guarantee principles and fault guarantee mechanisms.

3.3 Scalability

Scalability is one of the most important principles embedded in *Trilobita*'s design. It ensures its adaptability to varying computational workloads within a distributed environment. The system's scalable capability is evident in multiple dimensions, including *Cluster Size Scalability*, *Single Machine Parallelism Scalability*, and *Graph Partition Scalability*.

3.3.1 Cluster Size Scalability

Cluster size in *Trilobita* is scaled by the user. By leveraging the adjustable parameter 'numOfWorker' users can effortlessly expand or contract the size of their clusters to align with the demands of their graph processing tasks. This cluster size scalability ensures optimal resource utilization and responsiveness to varying workloads.

In addition, *Trilobita* exhibits automated resilience in facing server failures by dynamically adjusting the cluster size without requiring user intervention. This autonomous adaptation ensures continuous operation and optimal resource utilization, further enhancing the system's scalability and fault tolerance.

3.3.2 Single Machine Parallelism Scalability

Trilobita's scalability extends to the parallelism within each worker server, allowing users to configure each server. With the “parallelism” parameter, users can adapt the level of parallelism in each worker to match the computational requirements of their tasks. This fine-grained control over single-machine parallelism scalability enhances *Trilobita*'s efficiency, enabling it to utilize the full potential of available resources on individual machines.

3.3.3 Graph Partition Scalability

Efficient graph partitioning is crucial for scalable graph processing. *Trilobita* provides users with Graph Partition Strategy APIs. Based on the characteristics of the input graph, users can define the most efficient graph partition strategy by considering each worker’s computation sources. This graph partition scalability is pivotal for handling diverse graph sizes and complexities, making *Trilobita* scalable for large graph processing tasks.

Overall, *Trilobita*'s scalability is characterized by its prowess in adapting to varying cluster sizes, optimizing single-machine parallelism, and implementing scalable graph partitioning strategies. These aspects collectively contribute to the system's ability to efficiently scale and handle varying workloads.

3.4 Fault Tolerance

In this section, we explore fault tolerance in *Trilobita*, addressing unexpected server failures during graph processing tasks. These faults can disrupt the normal flow of distributed computation, potentially leading to data inconsistencies or processing failures. *Trilobita* distinguishes between two primary failure types: Master Failure, when the master server becomes unexpectedly unavailable, and Worker Failure, the unexpected cessation of one or more worker servers responsible for task execution. The subsequent discussions outline *Trilobita*'s fault resilience, covering master and worker failure scenarios, along with the fault handling mechanisms and implementation details of the heartbeat mechanism and snapshot. These mechanisms ensure the system's resilience, guaranteeing execution continuity in distributed environments.

3.4.1 Definition and Types of Faults

In the context of *Trilobita*, a fault is defined as ***any unexpected failure of a server in the middle of executing a task***. In a distributed system, faults can disrupt the normal flow of graph processing tasks and potentially lead to data inconsistencies or processing failures. To ensure robustness and continuity in the face of such faults, *Trilobita* distinguishes between two primary types of failures:

1. *Master Failure*: A Master Failure occurs when the master server becomes unexpectedly unavailable during the execution of graph processing tasks.
2. *Worker Failure*: A Worker Failure refers to the unexpected cessation of failure of one or more worker servers responsible for executing specific parts of the graph distributed by the master.

3.4.2 Fault Resilience

The objective of incorporating fault tolerance in *Trilobita* is to ensure the system's resilience in the face of faults, thereby promoting continuous and reliable graph processing in a distributed environment. Following this objective, the fault resilience in *Trilobita* is two-fold:

1. *Master Failure Resilience*: *Trilobita* adopts a “backup” approach to handle master failures by introducing multiple replicas into the distributed system. In the event of a master failure, an election process utilizing the bully algorithm is initiated. This process ensures the transition to a designated replica, which assumes the role of the master and takes over the coordination of graph processing tasks.

2. *Worker Failure Resilience*: To address worker failures, *Trilobita* implements a heartbeat mechanism at the master server to assess the state of worker servers. Upon detecting a worker failure, the master initiates a graph repartition and task re-execution process.

3.4.3 Implementations

In this section, we introduce the implementation of the heartbeat mechanism and snapshot, which are two primary components in our fault-handling process.

1. *Heartbeat Mechanism*: Servers in the system periodically emit their heartbeat signals. The master server actively listens to heartbeats from workers to ensure their liveness. In case any worker fails, the master will initiate a worker-faulty handling process, which repartitions the graph and restarts the superstep based on the most recent snapshot. Meanwhile, heartbeat exchanges occur between the master and its replicas. When the master encounters a failure, a master replica will promptly detect it, and one replica will be elected by the bully algorithm to take over the master position. In addition, when a failed worker rejoins the group, its heartbeat will be recaptured by the master. The master will therefore repartition the graph.

2. *Snapshot*: Periodically, the master server initiates a request for all workers to transmit their respective local vertex values to the master. After receiving, the master proceeds to update the graph stored within its local memory based on the received vertex values from workers. The updated graph is then encapsulated into a snapshot, which is stored in the master's persistent memory and synchronized across master replicas. When a master failure happens, the new master will roll back the system’s states to the most recent snapshot.

Together, these fault-handling mechanisms enhance *Trilobita*’s resilience in facing unexpected faults. The heartbeat mechanism, coupled with the periodic snapshot updates, underscores the system's ability to adapt to different conditions and recover gracefully from failures.

3.5 Limitations

In *Trilobita*, we assume the cluster size is fixed after the execution starts. This indicates that users cannot resize the cluster or add more workers to the cluster during the execution period. Dynamically resizing the cluster requires fine-grained management in server-level communications. We have started tackling this problem and, if you check our source code, we have implemented a *TrilobitaCluster* module to solve this problem. However, due to the challenging workload, this additional feature is still unstable. Another major limitation of *Trilobita* is the contention of communication is very high. This is a side-effect of our implementation of server-to-server communications. We detailedly discuss this problem in Section 4.3. Due to its significant complexity, we leave it as future work.

4 EVALUATION

In this section, we conduct a detailed experimental evaluation of *Trilobita*. Focusing on its correctness, scalability, and performance across various scenarios. We made the following key observations.

- *Trilobita* showcases robust fault tolerance capabilities and is adept at effectively managing diverse types of faults. This resilience ensures the system maintains data consistency and correctness, demonstrating its reliability throughout diverse executions.
- *Trilobita* exhibits the capability to process graphs in a distributed environment, showcasing high scalability in both the number of workers and parallelism settings. The system adapts to varying computational resources and is able to handle tasks across a diverse number of workers and parallelism configurations.
- *Trilobita* is more efficient in handling cases that involve large graphs and high computation complexity, especially in graph structures with vertices arranged in a fan-shaped manner.

We performed all experiments using our personal laptops with the Single-Source Shortest Paths algorithm, configuring the system with 1 master, 1 master replica, and 3 workers, each with a parallelism setting of 4 during execution. To demonstrate the correctness and efficiency of the distributed system, we further implemented a single-machine-based graph exploring system *Flucta*[2] to run the same tasks locally. By comparing the computation results of both, we proved *Trilobita*'s correctness in execution.

4.1 Correctness Evaluation

As discussed in section 3.2, by following the critical design of *Pregel*, we ensure that *Trilobita* is robust and consistent under both non-faulty and faulty conditions. The correctness is evaluated in terms of the final result. Since the Single-Source Shortest Paths algorithm is deterministic, it consistently converges to identical values in every execution. For all experiments in this section, we initiate the algorithm with three workers, one primary master, and one master replica.

4.1.1 No fault

By running the task in a no-fault environment, we observe that the final output is the same as the one that the *Flucta* produces, thus justifying its correctness in the no-fault environment.

4.1.1 Master Failure

We start the task normally and then terminate the execution of the primary master. After failing to detect the heartbeat from the primary, the replica loads the latest snapshot and continues the execution from that point. We observe that the final output is still as same as the correct result, which indicates the system's capability to recover from a previous superstep.

4.1.1 Worker Failure

Similar to the previous case, we start the algorithm normally and then terminate the execution of one of the workers. Unable to detect the failed worker's heartbeat, the primary master repartitions the graph and resumes the execution process with the rest of the workers. We observe that the final output is no different from the correct result, which indicates that the master is able to handle worker failure scenarios.

Table I shows our results of three cases in terms of running time and correctness. Our results demonstrate the correctness of *Trilobita* across all tested scenarios. Algorithmic outputs were consistent with expectations, and the system exhibited resilience to faults, recovering gracefully without compromising correctness.

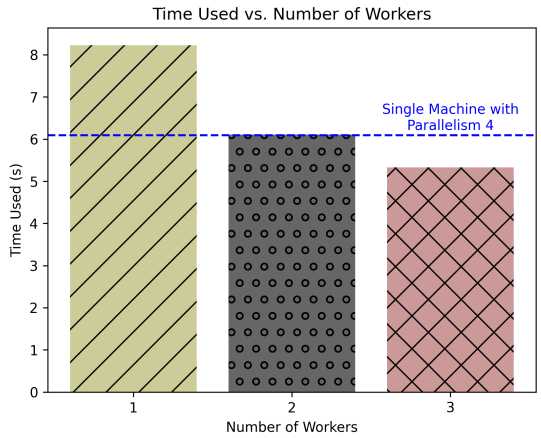
	Graph 1 (*s*)	**Graph 2 (*s*)**	**Correct Vertices**
No Fault	30.92	5.33	100%
Worker Failure	31.59	9.50	100%
Master Failure	29.19	8.87	100%

Table I: Performance Metrics Under Failures: The total execution time (in seconds) and the percentage of correct final vertex values under non-faulty and faulty conditions with 3 workers and 1 master. Here Graph 1 and Graph 2 refer to the deep graph and wide graph that we explain in section 4.3.3.

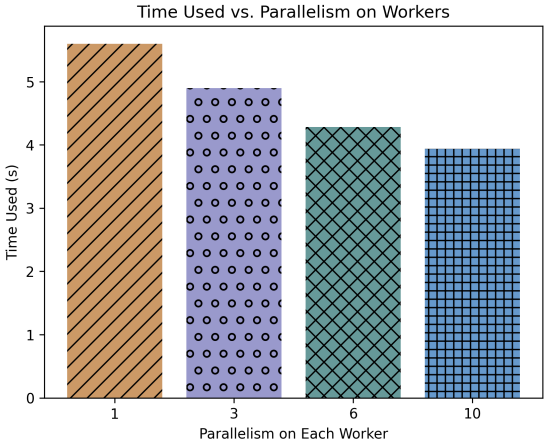
4.2 Scalability Evaluation

Following the discussion in Section 3.3, we evaluate the scalability of the system from two dimensions: the number of workers and parallelism.

We first vary the number of workers from 1 to 3, comparing the total execution time against the scenario of a single machine. As depicted in Figure 3, the total execution time exhibits a trend of decrease with an increasing number of workers. Notably, when the number of workers surpasses 2, *Trilobita* demonstrates superior performance over a Single Machine in the Shortest Path Task. We then vary the parallelism on workers and record the corresponding execution times. The results are shown in Figure 4. There are two key observations. Firstly, the total time used decreases as the number of workers increases. This indicates that the overhead caused by communication and synchronization can be further eliminated when the number of workers in the cluster grows larger. Secondly, the execution efficiency increases when the parallelism on each worker increases.



[Figure 3]: Effect of Number of Workers



[Figure 4]: Effect of Worker Parallelism

These observations underscore *Trilobita*'s scalability in executing tasks in a distributed environment. Through adjustments in the number of workers and parallelism settings, *Trilobita* can adapt to diverse workloads and computational resources.

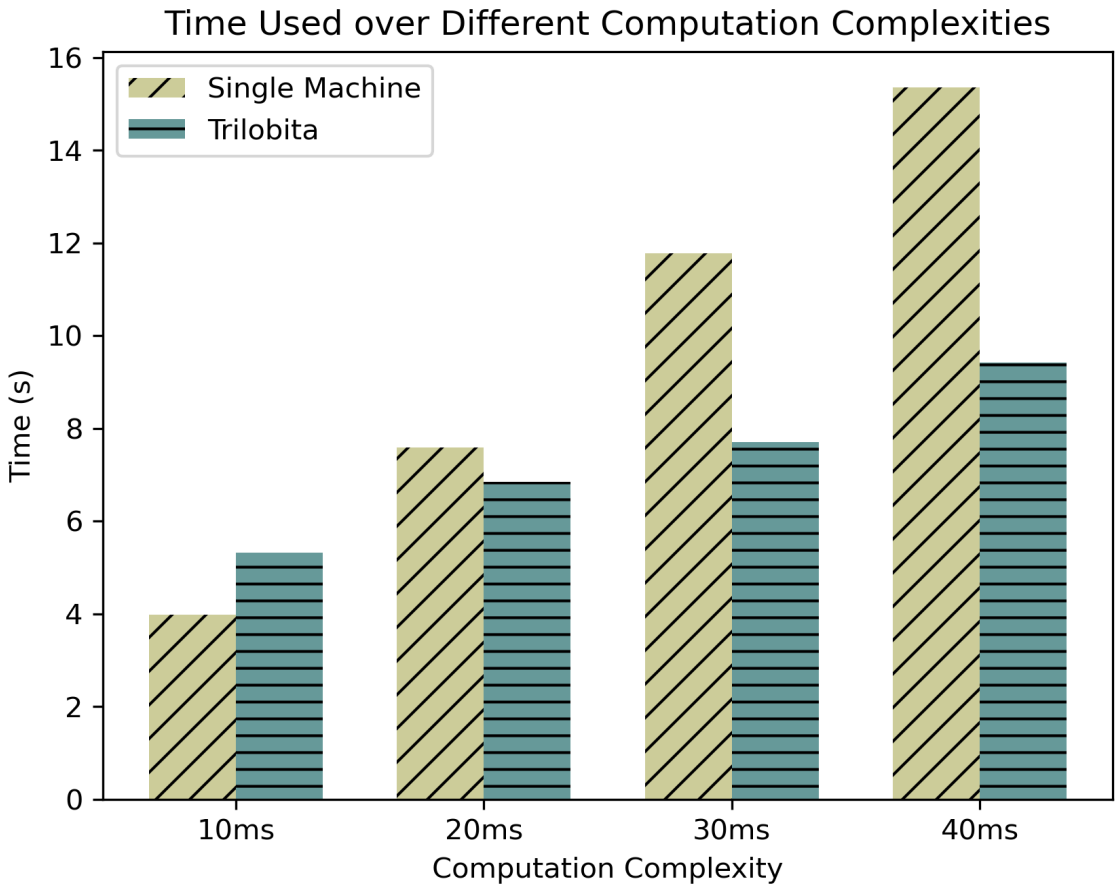
4.3 Performance Evaluation

We conducted some experiments to evaluate *Trilobita*'s performance in dealing with different workloads. We identified a few factors here, including the vertex task's computation complexity, the scale of the graph, and the depth of the graph.

4.3.1 Computation Complexity

In the previous experiment, we noticed that our one-worker case ran slower than a single machine. To this end, we identified several potential factors. Firstly, the reliance on Kafka for message communication introduced a significant latency overhead. The publish-subscribe model implemented by Kafka, while ensuring fault tolerance and scalability, incurred communication delays that impacted overall system responsiveness. Secondly, the inherent nature of the Pregel model, involving superstep synchronization, led to waiting periods between successive computation steps, especially in scenarios with complex graph structures. Thirdly, the vertex compute complexity in our experiments is relatively low, leading to the computation time being dwarfed by the time spent in inter-vertex communication.

To verify the third assumption, we modified the compute time of each vertex. As illustrated in Figure 5 and Table II, when the computation complexity increases, the portion of waiting time and messaging time significantly decreases. In addition, Figure 5 further demonstrates *Trilobita*'s superiority in processing complex tasks compared with a single machine.



[Figure 5]: Effect of Computation Complexity

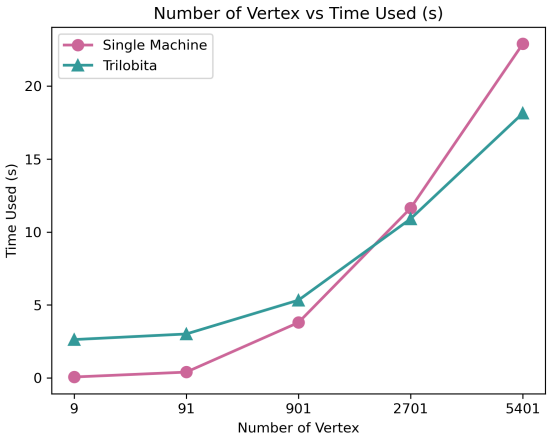
	Low Computation Complexity	High Computation Complexity
Distribution	15.8**%**	0.4%
Compute	8.7**%**	91.6%
Messaging	28.0**%**	3.5%
Waiting	47.5%	4.5%

Table II: Performance Metrics Breakdown: The proportion of time used in distributing messages, computing values, messaging, and waiting within a superstep. The time is measured to be the mean of all supersteps.

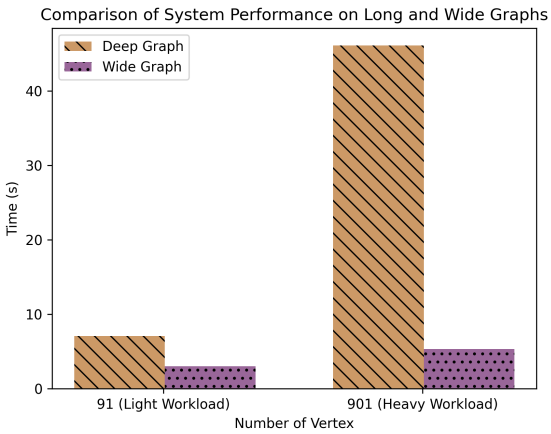
4.3.2 Graph Scale

To investigate the impact of graph scale on execution time, Figure 6 illustrates the runtimes of the algorithm with varying numbers of vertices in the graph. As the graph scale grows, the runtime on a single machine experiences a notable increase, contrasting with *Trilobita*'s consistent and moderate pace in time increment. This observation highlights *Trilobita*'s robust performance when processing graphs with large scales.

4.3.3 Evaluation of Different Graph Types



[Figure 6]: Effect of Workload



[Figure 7]: Effect of Graph Depth

During the experimentation with the shortest path algorithm, we observed significant performance differences in *Trilobita* when dealing with graphs with different depths. Deep graphs refer to those where all vertices form a graph in a nearly linear fashion. Wide graphs, on the other hand, involve vertices arranged in a fan-shaped manner. It is evident from the graphs that the wide-type graphs consistently exhibit faster processing, whether with fewer or more vertices, and the performance gap becomes more pronounced as the number of vertices increases.

Our understanding of this phenomenon is that wide graphs have fewer Supersteps, while each superstep involves handling multiple tasks, effectively utilizing the resources of each worker.

Conversely, deep graphs, while handling fewer tasks in each superstep, require a higher number of supersteps to fully explore the graph, and each superstep does not fully leverage the resources of each worker. Therefore, *Trilobita* performs better when dealing with wide graphs compared to deep graphs.

5 CONCLUSION

Large-graph processing in a distributed environment has been a major concern in both industrial and research settings. In this work, we developed *Trilobita*, a Pregel-like distributed graph processing system that is able to handle large-scale graph processing tasks in a distributed environment. Following our design principles, this system is divided into three layers, including Core, Engine, and Runtime. Complementing this architecture, we enriched *Trilobita* with a set of APIs, enhancing its adaptability and scalability. Finally, we conduct a series of experiments to evaluate *Trilobita*'s correctness, scalability, and performance. We confirm that it delivers robust correctness guarantees and fault tolerance alongside high scalability contributed by the system's inner design and user-definable classes. As we look forward, future work could focus on refining cluster-level features and optimizing the server-level communication mechanisms for enhancements.

6 ADDITIONAL INFORMATION

We provide some reference links that might help you explore and understand our system.

[1]. *Trilobita* source code and documentation: https://github.com/TsukiSky/Trilobita

[2]. *Flucta* source code: https://github.com/TsukiSky/Flucta

[3]. *Pregel* paper: https://15799.courses.cs.cmu.edu/fall2013/static/papers/p135-malewicz.pdf