GraphMa: Towards new Models for Pipeline-Oriented Computation on Graphs

Daniel Thilo Schroeder
SINTEF
Norway
daniel.t.schroeder@sintef.no

Tobias Herb
Germany
tobias.herb@gmx.de

Brian Elvesæter
SINTEF
Norway
brian.elvesater@sintef.no

Dumitru Roman
SINTEF
Norway
dumitru.roman@sintef.no

ABSTRACT

This paper presents GraphMa, a framework aimed at enhancing pipeline-oriented computation for graph processing. GraphMa integrates the principles of pipeline computation with graph processing methodologies to provide a structured approach for analyzing and processing graph data. The framework defines a series of computational abstractions, including computation as type, higher-order traversal, and directed data-transfer, which collectively facilitate the decomposition of graph operations into modular functions. These functions can be composed into pipelines, supporting the systematic development of graph algorithms. For this paper, our focus lies in particular on the capability to implement the well-established computational models for graph processing within the proposed framework. In addition, the paper discusses the design of GraphMa, its computational models, and the implementation details that illustrate the framework's application to graph processing tasks.

CCS CONCEPTS

• **Computer systems organization → Pipeline computing**.

KEYWORDS

Graph processing, Pipeline-oriented computation, Graph data, Graph operations

ACM Reference Format:
Daniel Thilo Schroeder, Tobias Herb, Brian Elvesæter, and Dumitru Roman. 2024. GraphMa: Towards new Models for Pipeline-Oriented Computation on Graphs. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24 Companion), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3629527.3652894

Tobias Herb and Daniel Thilo Schroeder contributed equally.

ICPE '24 Companion, May 7–11, 2024, London, United Kingdom

ACM ISBN 979-8-4007-0445-1/24/05
https://doi.org/10.1145/3629527.3652894

1 INTRODUCTION

Graphs, with their intricate structures and complex relationships, are an integral part of the world around us, playing a key role in various domains ranging from social networks to biological systems. Historically, graph processing has been fundamental in applications such as network analysis, e.g. shortest path computation [12, 22], for mapping services [5], analysis of social networks [13, 21] or even feature extraction [17]. These traditional uses have paved the way for more advanced applications.

In recent times, we have witnessed the emergence of graph processing in areas like recommendation systems [3, 4, 7, 20], fraud detection [11], and complex interaction mapping in bioinformatics. Moreover, the increasing availability and collection of large datasets have significantly influenced the size and complexity of graphs [16]. These large-scale graphs present unique challenges and opportunities for processing and analysis.

In response to these challenges, distributed graph processing has gained prominence [2]. This shift from traditional, localized graph processing to distributed methods addresses the need for scalability and efficiency in handling vast and more and more complex graph structures. Frameworks like Apache Giraph [10], Google's Pregel [9] or the Apache TinkerPop [14] project have been instrumental in this transition. They offer powerful, general-purpose solutions for distributed graph processing, enabling easier implementation of algorithms tailored to specific graph-related problems.

Building upon the foundational aspects of graph processing, the application of concepts like immutability or modularity, well-known in functional programming, offer a promising approach for constructing graph processing pipelines. The congruence between these concepts and graph processing is rooted in several advantages, which are particularly beneficial for handling the challenges posed by graph data.

In this context, modularity is the decomposition of complex problems into smaller, reusable functions. This approach mirrors the inductive nature of many graph algorithms and processing workflows, where operations are independent yet interrelated. In graph processing, this translates to the ability to encapsulate operations like traversal, filtering, and transformation into discrete functions, which can then be composed to address more complex graph-related problems. Such modularity not only fosters code reusability but also simplifies the process of constructing and maintaining graph

algorithms and even beyond, the composition of a multitude of graph algorithms.

Graph processing, especially when dealing with large datasets, can greatly benefit from parallel and concurrent execution. Reducing or eliminating shared mutable states, simplifies the management and reasoning of parallelism and concurrency. This is crucial for optimizing the performance of graph algorithms, allowing them to handle the computational demands of large-scale graph analysis. The immutability and stateless nature embedded in the concepts of functional programming make it inherently suited for these tasks.

In this paper, we outline the fundamental principles that underlay our concept for pipeline-oriented computation in general. Building on this we propose and discuss GraphMa, a collection of ideas and preliminary implementations extending the ideas around general pipeline-oriented computation towards graph processing. For this paper our focus lies in particular on the capability to implement the well-established computational models for graph processing (see Section 2) within the proposed framework (see Section 3).

2 BACKGROUND: COMPUTATIONAL MODELS FOR GRAPH PROCESSING

The landscape of graph processing is rich and varied, encompassing a range of computational units and models each designed to optimize different aspects of graph analysis. At the heart of these models is the goal to efficiently process and analyze data structured in graphs. Based on [2] we would like to exercise an overview of the primary computational models that have shaped modern graph processing, highlighting their foundations, operations, and the challenges they address.

- *Vertex-Centric (TLAV) Model:* Pioneered by Google's Pregel [9] and further extended by Apache Giraph [1], the Vertex-Centric model places the vertex at the center of computation. In this model, each vertex independently executes the same function, processing incoming information, potentially updating its state, and then communicating with other vertices through its edges. This approach allows for high levels of parallelism as each vertex operates in isolation, yet collaboratively contributes to the graph's overall computation.
- *Superstep Paradigm:* This execution model, integral to Vertex-Centric processing, organizes computation into a series of global steps known as supersteps. During a superstep, vertices concurrently execute a specified function, after which they engage in communication by sending messages to vertices that will be active in the subsequent superstep. This synchronized execution and communication phase structure not only facilitates easier reasoning about the computational process and thus simplifies the programming of distributed graph algorithms. The paradigm is briefly described in [2]. We also recommend read [19] and [9].
- *Scatter-Gather Model:* This model splits the process of message handling into two distinct phases: scattering, where vertices send out messages, and gathering, where messages are collected and state updates are aggregated. By clearly distinguishing between these phases, the Scatter-Gather model [18] provides a structured approach to handling vertex communication, facilitating more organized data processing flows.
- *Gather-Apply-Scatter Model:* Introduced by PowerGraph [6] adresses the challenge of computational load imbalance, especially in graphs with power-law distributions, the Gather-Apply-Scatter model decomposes vertex operations into three phases: gather information from neighboring vertices, apply a function to update the vertex's state, and scatter results to influence neighboring vertices in the next cycle.
- *Edge-Centric Model:* Offering a different perspective, the Edge-Centric model [15] focuses computation on graph edges rather than vertices. This model, exemplified by X-Stream and Chaos, is particularly effective in scenarios where edge-based computations are predominant. This model optimizes the use of secondary storage and network communication, making it suitable for processing very large graphs that do not fit into memory.
- *Sub-graph-Centric Model:* By concentrating on sub-graphs, either partition-centric within a physical partition or neighbourhood-centric allowing for shared state updates, this model [8] aims to reduce communication overheads. This approach is especially beneficial in distributed environments where minimizing inter-node communication can significantly enhance performance.
- *MEGA Model:* Specifically designed for machine learning applications on graphs, the MEGA model introduced by Tux2 [23] focuses on edge-level computations with functions such as Exchange, Apply, and Global Sync. These functions facilitate detailed manipulation of graph structure and values, supporting sophisticated machine learning algorithms on graph data.

3 A CONCEPTUAL FRAMEWORK TO PIPELINE-ORIENTED COMPUTATION

This section presents a comprehensive overview of our novel computation model designed for pipeline-oriented data processing in general. It is only in Section 4 when we discuss how to apply this framework as the basis to implement computational models for graph processing. Our proposed model integrates functional programming paradigms with object-oriented design principles to create a versatile framework capable of addressing complex data processing requirements. It is structured around a series of interconnected layers, each contributing to a cohesive and flexible architecture that facilitates the development of data processing pipelines. These layers include *Computation as Type*, *Higher-order Traversal Abstraction*, *Directed Value-Transfer Protocol*, *Operator Model*, and finally the Pipeline Abstraction. We begin to introduce the first Layer *Computation as Type*.

3.1 Computation as Type

The foundation of our model is the concept of Computation as Type, which posits computation units as first-class entities encapsulated by a `Compute` interface. This interface is defined as a function accepting a value of type `T` and performing operations, potentially

with side effects. This foundational abstraction underpins the architecture for creating pipeline stages, enabling data processing that is both type-safe and modular.

3.2 Higher-order Traversal Abstraction

The second layer, Higher-order Traversal Abstraction is the component designed to oversee data access and processing in an abstract manner. It outlines the methods for navigating through different data sources, thereby enabling versatile and effective data manipulation. In this section, we explore the fundamental elements of this abstraction, detailing their organizational framework and how they interact.

3.2.1 Structural Composition and Behavioral Interaction.

Higher-order Traversal Primitives: At the core of our Traversal Abstraction are the higher-order traversal primitives, which are instrumental in defining the manner in which data is accessed and iterated over. These primitives are characterized by their ability to abstract away the specifics of data sources and access mechanisms, providing a unified interface for data traversal. Key characteristics include:

- *Sequential Access*: An abstraction layer that decouples the traversal mechanism from the data source's physical representation while allowing for sequential access to data, enabling iteration over data sources like containers, IO channels, or generator functions.
- *Computation Integration*: An important feature is the integration with first-order computation units, allowing traversed values to be processed in a seamless and flexible manner. This is achieved through second-order functions that pass each traversed value to a specified computation unit (`Compute<T>`).
- *Traversal Strategies*:
  - *Single-step Traversal (*`TryNext`*)*: Processes data one item at a time, affording precise control over the iteration and enabling fine-grained data manipulation.
  - *Bulk Traversal (*`ForNext`*)*: Optimizes data processing by handling batches of data, streamlining the traversal process and improving efficiency.
  - *Continuation-controlled Bulk Traversal (*`WhileNext`*)*: Introduces a continuation-passing style for bulk processing, offering dynamic control over the traversal logic based on runtime conditions. This strategy is particularly notable for its use of the `Continuation` interface, which provides a mechanism for halting or altering the course of computation in response to specific criteria.

Traverser Abstraction in a nutshell:

- *Computation Carrier*: The `Traverser` emerges as the central figure in this abstraction, acting as the carrier for the computation across data sets. It encapsulates the higher-order traversal primitives, serving as the execution context for data processing operations.
- *Unified Control Flow Patterns*: By housing different traversal strategies within a coherent framework, the `Traverser` harmonizes flexibility with control. It offers a spectrum of control flow patterns, from granular, step-by-step data processing to more coarse-grained, bulk handling techniques.
- *Seamless Interaction and Modularity*: The delineation of traversal strategies into distinct components not only clarifies the traversal abstraction but also enhances the system's modularity. This separation allows for the extension and customization of traversal behaviors to accommodate specific processing requirements, fostering reusability and adaptability.
- *Foundation for Advanced Data Processing*: The integration of traversal primitives within a unified `Traverser` environment provides a robust foundation for implementing sophisticated data processing strategies. This design carefully balances the need for complex control flow mechanisms with the desire for a clear, modular architectural structure.

By abstracting the intricacies of data traversal and offering a suite of customizable traversal strategies, the Traverser layer stands as a cornerstone of the proposed computation model. It exemplifies the framework's capacity to facilitate advanced data manipulation techniques.

3.3 Directed Data-Transfer Protocol

We introduce a refined conceptual model for pipeline-oriented computation, anchored by the Directed Value-Transfer Protocol. This model emphasizes the seamless management and transfer of data across computational stages, aligning with the principles of functional programming and type-centric design philosophies. Our model is distinguished by its lifecycle-aware architecture and the explicit delineation of data producer and consumer roles, facilitating a structured yet flexible approach to constructing computation pipelines.

3.3.1 Architectural Foundations. At the heart of our model lies the `Compute<T>` interface, a fundamental abstraction representing a unit of computation. It encapsulates the notion that computations are first-class entities, capable of accepting input and executing operations in a type-safe manner. Building upon this, our design introduces a hierarchical structure aimed at enhancing data transfer efficiency and lifecycle management:

- *Lifecycle Integration*: The `Transfer.Lifecycle` interface introduces a dual-phase lifecycle management protocol with `open()` and `close()` methods. This protocol ensures the acquisition and release of resources are handled gracefully, enhancing the robustness of the computation chain.
- *Role-Specific Abstractions*: Our model defines two important interfaces, `Transfer.Port<T>` and `Transfer.Pipe<T>`, to represent the roles of data producers and consumers, respectively. This distinction not only clarifies the data flow directionality, but also enriches the model with the capability to handle complex data processing scenarios.

3.3.2 Operational Dynamics. The Directed Value-Transfer Protocol underpins an interaction framework:

- *Managed Data Flow*: The explicit lifecycle management embedded within the data transfer interfaces ensures that each stage of the computation pipeline is initialized and terminated appropriately, promoting efficient resource utilization.

- *Directed Transfer Mechanism*: By segregating data transfer roles into `Port` and `Pipe`, our model ensures a clear and efficient directionality of data flow. This segregation allows for the optimization of data transmission mechanisms, catering to both synchronous and asynchronous processing needs.
- *Flexible Computation Chains*: The extension of `Pipe<T>` to potentially act as both consumer and producer underscores the model's versatility. It supports the construction of intricate multi-stage processing pipelines, enabling data to be transformed progressively through successive computational units.

3.3.3 Computational Chain Composition. A key feature of our model is the `LazyChain<I,O>` interface, which symbolizes the essence of pipeline-oriented computation through the contravariant composition of computation units. This interface facilitates the dynamic assembly of computational stages, allowing for the efficient transformation and transfer of data across the pipeline:

- *Enhanced Modularity*: The `LazyChain` interface exemplifies the model's commitment to modular and reusable design principles. It allows for the flexible chaining of computational units, ensuring that complex data processing tasks can be decomposed into manageable, composable segments.

The Directed Value-Transfer Protocol, as conceptualized in our pipeline-oriented computation model, represents a sophisticated framework for data processing. It marries the principles of lifecycle management, type safety, and functional programming to offer a robust and flexible solution for constructing complex computational pipelines. Through this model, we aim to provide a scalable and efficient framework for addressing the diverse challenges of modern data processing tasks, reaffirming the potential of functional patterns in the realm of object-oriented programming languages like Java.

3.4 Operator Model

The Operator Model represents the quintessential fifth layer within our innovative pipeline-oriented computational framework, meticulously crafted to underpin the construction and orchestration of data processing pipelines. This model introduces a sophisticated suite of computational constructs, pivotal for the lifecycle management of operators and the nuanced handling of their states, thereby facilitating a broad spectrum of data processing operations. Herein, we integrate and refine the abstract conceptualization of the Operator Model, emphasizing its core constructs, their structural interplay, and the pivotal role of terminal operators in concluding data processing tasks.

3.4.1 Core Constructs and Structural Composition.

Operator Protocol. The `Operator<T>` abstraction stands as the cornerstone of the Operator Model, extending Transfer.Lifecycle to underscore its essential role in managing the lifecycle and state of operations within pipelines. This interface is instrumental for:

- *State Management*: It allows for the encapsulation of stateful computations, enabling operators to maintain and manipulate local state through the `localState()` method.

Transducer. The `Transducer<I, O>` abstraction, serving as the backbone for intermediate operators, embodies the transformational logic necessary for processing and relaying data through various stages of the pipeline. Its design is focused on:

- *Transformation and Lazy Computation*: Facilitating the lazy transformation of data, thus acting as a critical bridge in the data flow across the pipeline.

Materializer. The `Materializer<T>` abstraction plays a crucial role in state materialization, especially in managing the transition of data states within pipelines through chunked buffers, enhancing the efficiency and organization of data processing workflows.

Terminal Operators. Terminal operators, categorized into Complete Terminal Operators and Partial Terminal Operators, mark the culmination of the pipeline's data processing journey. They are distinguished by their evaluation strategies:

- *Complete Terminal Operators* process the entirety of input data, embodying exhaustive data analysis or transformation.
- *Partial Terminal Operators* facilitate early termination of processing based on specific conditions, optimizing performance through lazy evaluation and early termination strategies.

3.4.2 Interaction Dynamics and Evaluation Strategies.

- *Lifecycle and State Management*: The Operator Model ensures meticulous lifecycle management across all operator types, harmonizing state management and data transformation processes. This integration is vital for the seamless flow and transformation of data across the pipeline.
- *Flexible Terminal Evaluation*: The differentiation in terminal operator strategies enhances the model's adaptability, allowing for both exhaustive data processing and efficient, condition-based evaluations. This flexibility ensures optimal performance and resource utilization, catering to a wide range of computational requirements.

The Operator Model emerges as a comprehensive and modular framework for pipeline construction, characterized by its advanced management of operator lifecycle, state, and terminal evaluation strategies. Through its well-structured abstractions—from transducers and materializers to the nuanced categorization of terminal operators—it lays a versatile and extensible foundation for domain-specific data processing operations. This model not only encapsulates the core principles of pipeline-oriented computation but also fosters adaptability and efficiency, ensuring its applicability across diverse data processing scenarios.

3.5 Pipeline Abstraction

The `Pipeline` abstraction is a pivotal construct within the novel pipeline-oriented computational model, representing the overarching framework that orchestrates the structured and stateful processing of data. This abstraction serves as a high-level blueprint for defining data processing flows, encapsulating the complexities of data transformation and transmission. The design principles underlying the `Pipeline` model prioritize modularity, flexibility, and clarity in constructing computational logic, thereby offering a robust platform for implementing sophisticated data processing mechanisms.

3.5.1 Overview of Pipeline Components.

- *Stage*: A `Stage`within the pipeline signifies a discrete processing unit, tasked with receiving input values, applying a specified operation, and producing output for the subsequent stage. It embodies the core functional element of the pipeline, enabling the definition and execution of transformation operations in a type-safe manner.
- *State*: The `State` component encapsulates stateful logic for data materialization, transforming or accumulating data as it traverses the pipeline. This aspect of the pipeline architecture facilitates the implementation of complex data processing semantics, allowing for the dynamic evolution of computation based on the flow of data.
- *Sink*: Serving as the terminal point of the pipeline, the `Sink` is responsible for consuming all processed data to produce a final outcome or effectuate a side operation. It marks the culmination of the pipeline's computational process, transitioning the abstract pipeline description into an actionable computation through the evaluation operator.

3.5.2 Structural Composition. The pipeline is conceptually structured as a series of computation steps, organized as `LazyChain` instances and interconnected via `Transfer.Pipe` objects. These steps converge at an `Operator.Terminal`, where the computed data is either transformed into a result or utilized to perform a side effect. The recursive type parameterization of the `Pipeline` interface ensures type safety across the processing stages, facilitating the seamless chaining of operations.

3.5.3 Interaction and Evaluation Strategies. The pipeline model embraces the principle of "laziness," deferring computations until absolutely necessary. This design choice enables the efficient assembly of an execution plan that outlines the data transformation process, from the source through to the sink. The plan encapsulates the requisite parameters for executing computations at each stage, culminating in a pipeline sink where the evaluation operator resides.

The evaluation operator, embodied by the `Evaluator`, allows for the implementation of various evaluation strategies:

- *Eager Evaluation*: Immediate execution of computations upon their invocation.
- *Lazy Evaluation*: Deferral of computations until required, encapsulated within a 'thunk' to capture the deferred computation.
- *Memoized Evaluation*: Computations are performed upon first access, with results cached for future reference.

This flexible evaluation framework, referred to as the 'Flow-Machine', grants granular control over the computation flow, enhancing resource utilization and potentially improving execution speed and memory efficiency depending on the use case scenario.

The `Pipeline` abstraction forms the crux of a sophisticated computational model designed to facilitate the structured and stateful processing of data. Through its modular composition, the pipeline model enables the construction of complex data processing flows with ease, offering a comprehensive framework for the implementation of diverse computational logic. This abstraction not only simplifies the development of data processing applications but also enriches the computational model with a flexible and powerful mechanism for data transformation and evaluation.

4 IMPLEMENTATION OF COMPUTATIONAL MODELS FOR GRAPH PROCESSING IN GRAPHMA

In this section we propose approaches on how to embed graph computation models into the higher-order pipeline model.

4.1 Vertex-Centric Embedding

Embedding the *Vertex-Centric Computation Model* (TLAV) into the higher-order pipeline model could leverage the strengths of both models to efficiently process graph-based data.

Below we give a concise overview of how this integration is currently structured and operates.

4.1.1 Structural Composition.

(1) **Compute<T>for Vertex Operations**: Vertices are encapsulated as `Compute<Vertex>` instances, where each vertex acts as an independent computational unit with its own state. This aligns with the Computation as Type principle, allowing vertices to process data and messages in a type-safe manner.
(2) **Traversal as Message Passing**: The Higher-order Traversal Abstraction is adapted to facilitate message passing between vertices. Each vertex employs traversal primitives to send and receive messages, abstracting the communication mechanism and ensuring flexibility in message dissemination strategies.
(3) **Directed Data-Transfer for Supersteps**: The Directed Data-Transfer Protocol orchestrates the execution of supersteps. A `Transfer.Pipe<Message>` interface manages the asynchronous delivery of messages between supersteps, ensuring that messages sent in one superstep are correctly queued for processing in the next.
(4) **Operator Model for Vertex Execution Logic**: The Operator Model is extended to define vertex execution logic within supersteps. `Operator<Vertex>` interfaces manage state transitions and message processing, supporting the iterative nature of vertex-centric computations.
(5) **Pipeline Abstraction for Graph Processing Flows**: The entire graph processing logic is encapsulated within a `Pipeline<Graph>` abstraction, orchestrating the flow of computation across supersteps. This pipeline integrates stages for message passing, vertex state updates, and global convergence checks.

4.1.2 Behavioral Interaction.

(1) **Iterative Pipeline Stages**: Each superstep is represented as a stage in the pipeline, with vertices operating in parallel to process incoming messages and update their states. The pipeline dynamically adapts to the iterative nature of the vertex-centric model, allowing for repeated execution of stages until a global stopping condition is met.
(2) **Dynamic Message Routing**: The pipeline utilizes the Higher-order Traversal Abstraction to dynamically route messages

between vertices. This enables efficient scatter-gather operations, optimizing the distribution and collection of messages across the graph.

(3) **Stateful Computation and Lifecycle Management**: The Directed Data-Transfer Protocol and Operator Model jointly manage the lifecycle of vertex computations. They ensure that vertex states are correctly initialized, updated, and finalized across supersteps, maintaining consistency and robustness in the computation.

(4) **Flexible Evaluation Strategies**: The Pipeline Abstraction supports flexible evaluation strategies for vertex-centric computations, allowing for both eager and lazy execution of supersteps. This flexibility aids in optimizing performance based on the specific characteristics of the graph and the computational workload.

(5) **Adaptation to Vertex-Centric Variations**: The pipeline model's modular design allows for easy adaptation to different vertex-centric variations (bulk synchronous parallel, asynchronous, edge-centric, mixed-mode). Specific components of the pipeline (e.g., message routing, state management) can be customized to reflect the desired computational semantics and performance characteristics.

4.1.3 Summary. By integrating the Vertex-Centric Computation Model within the higher-order pipeline model, this approach provides a structured yet flexible platform for graph processing.

It combines the modularity, type safety, and functional programming strengths of the pipeline-oriented model with the intuitive, scalable, and vertex-focused computation of the vertex-centric model.

This integration not only enhances the expressiveness and efficiency of graph processing tasks but also leverages the parallel execution capabilities inherent in distributed computing environments.

4.2 Edge-Centric Embedding

Embedding the *Edge-Centric Computation Model* (TLAE) into the higher-order pipeline model involves focusing on the relationships and interactions between vertices, with edges acting as the primary conduits of computation and communication.

Below we give a concise overview of how such an integration could be structured and operate.

4.2.1 Structural Composition.

(1) **Compute<Edge>for Edge Operations**: Edges are represented as `Compute<Edge>` instances, where each edge acts as an independent computational unit capable of accessing and modifying the data of its connected vertices as well as its own properties. This encapsulation aligns with the Computation as Type principle, facilitating type-safe edge operations.

(2) **Directed Data-Transfer for Edge-Vertex Communication**: A `Transfer.Pipe<Message>` interface is utilized for edge-to-vertex message passing, managing the asynchronous exchange of messages. This supports direct communication between edges and vertices, allowing edges to send messages that influence vertex state and behavior.

(3) **Operator Model for Edge Execution Logic**: The Operator Model is adapted to define the logic of edge computations within supersteps. `Operator<Edge>` interfaces handle the processing tasks of edges, including state transitions based on both edge properties and received vertex messages.

(4) **Pipeline Abstraction for Edge-Centric Flows**: The graph processing logic, focusing on edge interactions, is encapsulated within a `Pipeline<Edge>` abstraction. This pipeline manages the stages of edge computation, message passing, and the integration of edge-induced vertex updates.

4.2.2 Behavioral Interaction.

(1) **Iterative Pipeline Stages for Edges**: Supersteps are represented as stages in the pipeline, with edges performing computation and message passing in parallel. This approach facilitates the autonomous operation of edges, allowing for the iterative processing of edge and vertex interactions across supersteps.

(2) **Dynamic Edge-to-Vertex Message Routing**: Utilizing the Directed Data-Transfer Protocol, the pipeline dynamically routes messages from edges to their connected vertices. This mechanism is crucial for enabling edges to influence vertex state and initiate vertex-level computations based on edge-centric logic.

(3) **Stateful Edge Computation and Lifecycle Management**: The integration of the Operator Model with the Directed Data-Transfer Protocol ensures that edge states are correctly managed throughout the computation lifecycle. This includes initialization, state updates based on incoming messages, and finalization, maintaining robustness and consistency in edge-centric computations.

(4) **Flexible Evaluation Strategies for Edge-Centric Operations**: The Pipeline Abstraction supports both eager and lazy execution strategies for edge-centric computations. This flexibility allows for performance optimization based on the graph's characteristics and the computational workload, enhancing the efficiency of edge-centric processing.

(5) **Adaptation to Edge-Centric Variations**: The modular design of the model facilitates easy adaptation to different edge-centric variations (pure edge-centric, vertex-augmented, hybrid). Specific pipeline components (e.g., message routing, edge computation logic) can be tailored to reflect the computational semantics and performance characteristics desired for each variation.

4.2.3 Summary. By embedding the Edge-Centric Computation Model within the higher-order pipeline model, this approach offers a structured platform for focusing on edge-based interactions and data flows in graph processing.

It leverages the modularity, type safety, and functional programming benefits of the pipeline-oriented model, along with the direct communication and computation capabilities inherent in edge-centric approaches.

This integration not only provides a powerful tool for addressing problems where edge relationships are paramount but also enhances the flexibility and performance of graph processing tasks in distributed computing environments.

4.3 Sub-Graph-Centric Embedding

Embedding the Sub-Graph-Centric Computation Model (TLAG) into the pipeline-oriented computation model focuses on leveraging cohesive groups within the graph structure, enhancing computational efficiency and expressiveness. This integration aims to exploit the locality and reduce communication needs by treating subgraphs as primary computational units.

Below we explain how such an integration could be structured and operate.

4.3.1 Structural Composition.

(1) **Compute<SubGraph>for Subgraph Operations**: Subgraphs are encapsulated as `Compute<SubGraph>` instances, treating each subgraph as an independent computational unit. This approach aligns with the Computation as Type principle, facilitating type-safe operations on subgraph data, including both its internal structure and interconnections.
(2) **Directed Data-Transfer for Subgraph Communication**: Utilizing the `Transfer.Pipe<Message>` interface, the model manages asynchronous message passing between subgraphs. This enables subgraphs to communicate, exchanging information and updates in a manner that respects the locality of data and computations.
(3) **Operator Model for Subgraph Execution Logic**: The Operator Model adapts to define subgraph-level computations. `Operator<SubGraph>` interfaces are responsible for executing computations that consider the subgraph's entire structural and relational context, supporting iterative execution patterns for dynamic state updates.
(4) **Pipeline Abstraction for Subgraph-Centric Processing**: The graph processing logic, centered around subgraph computations, is encapsulated within a `Pipeline<SubGraph>` abstraction. This pipeline coordinates the stages of subgraph computation, communication, and integration of updates across the larger graph structure.

4.3.2 Behavioral Interaction.

(1) **Iterative Pipeline Stages for Subgraphs**: Each pipeline stage corresponds to a superstep in the subgraph-centric computation, allowing subgraphs to process information and interact in parallel. This iterative approach facilitates the dynamic exchange of information and updates across subgraphs, maintaining the model's emphasis on locality and reduced communication overhead.
(2) **Dynamic Subgraph-to-Subgraph Communication**: The pipeline uses the Directed Data-Transfer Protocol to enable efficient and localized message passing between subgraphs. This setup is crucial for maintaining data locality and reducing communication overhead, particularly for algorithms that benefit from intensive local interactions.
(3) **Stateful Subgraph Computation and Lifecycle Management**: Integrating the Operator Model with the Directed Data-Transfer Protocol ensures that subgraph computations are managed effectively throughout their lifecycle. This includes initialization, iterative processing based on structural and relational context, and finalization, ensuring consistency and robustness in subgraph-centric computations.
(4) **Flexible Evaluation Strategies for Subgraph-Centric Operations**: The Pipeline Abstraction supports various evaluation strategies, including eager and lazy execution, tailored to the computational needs of subgraph-centric processing. This flexibility allows for optimization of performance based on the graph's structure and the computational workload, enhancing the efficiency of processing within and across subgraphs.
(5) **Adaptation to Subgraph-Centric Variations**: The model's modular design facilitates easy adaptation to different subgraph-centric variations (TLAG, graph-centric, neighbourhood-centric, hybrid). Specific pipeline components can be customized to reflect the computational semantics and performance characteristics desired for each variation, ensuring that the approach is tailored to the specific requirements of the problem at hand.

4.3.3 Summary. By embedding the Sub-Graph-Centric Computation Model within the higher-order pipeline model, this approach offers a structured yet flexible platform for focusing on computations within cohesive subgraph units.

It leverages the strengths of the pipeline-oriented model – modularity, type safety, and functional programming benefits – along with the enhanced locality, reduced communication needs, and expressiveness of the subgraph-centric approach.

This integration not only provides a powerful mechanism for addressing graph processing challenges that benefit from subgraph-level focus but also enriches the computational model with advanced capabilities for handling complex patterns and relationships in large-scale graph data.

5 CONCLUSION

In this paper, we introduced GraphMa, a collection of ideas and preliminary implementations extending the ideas around general pipeline-oriented computation towards graph processing. We argued that GraphMa's architecture, which merges pipeline computation principles with graph processing techniques, provides a structured method for constructing and executing graph algorithms. Through the introduction of computational abstractions such as computation as type, higher-order traversal abstraction, and directed data-transfer protocol, GraphMa enables the decomposition of complex graph operations into modular, composable functions.

Furthermore, we have qualitatively explored the potential integration of well-established computational models for graph processing within the GraphMa framework. This exploration has highlighted the framework's inherent flexibility and the theoretical effectiveness of such an integration. By detailing how these computational models could align with GraphMa's pipeline-oriented architecture, we have shed light on the framework's potential to facilitate and enhance the development and execution of graph processing tasks.

Looking ahead, we anticipate that GraphMa will serve as a valuable tool for researchers and practitioners in the field of graph processing, offering a scalable and modular approach to algorithm development. Future work will involve extending GraphMa's capabilities, exploring its application to a broader range of graph processing scenarios, and evaluating its performance in comparison

to existing frameworks. Our goal is to continue refining GraphMa, ensuring that it remains a robust and adaptable framework capable of addressing the evolving challenges in graph processing.

ACKNOWLEDGMENT

This work has been funded by the Graph-Massivizer project, which receives funding from the Horizon Europe research and innovation program of the European Union under grant agreement No 101093202.

REFERENCES

[1] Avery Ching. 2013. Scaling apache giraph to a trillion edges.
[2] Miguel E Coimbra, Alexandre P Francisco, and Luís Veiga. 2021. An analysis of the graph processing landscape. *journal of Big Data* 8, 1 (2021), 1–41.
[3] Xiaohui Cui, Xiaolong Qu, Dongmei Li, Yu Yang, Yuxun Li, and Xiaoping Zhang. 2023. MKGCN: Multi-Modal Knowledge Graph Convolutional Network for Music Recommender Systems. *Electronics* 12, 12 (2023), 2688.
[4] Leyan Deng, Defu Lian, Chenwang Wu, and Enhong Chen. 2022. Graph Convolution Network based Recommender Systems: Learning Guarantee and Item Mixture Powered Strategy. *Advances in Neural Information Processing Systems* 35 (2022), 3900–3912.
[5] Lau Nguyen Dinh. 2024. The MapReduce based approach to improve the all-pair shortest path computation. *International Journal of Advanced Science and Computer Applications* 3, 1 (2024), 55–64.
[6] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, Hollywood, CA, USA, 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez
[7] Elvin Isufi, Matteo Pocchiari, and Alan Hanjalic. 2021. Accuracy-diversity trade-off in recommender systems via graph convolutions. *Information Processing & Management* 58, 2 (2021), 102459.
[8] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. 2017. High-level programming abstractions for distributed graph processing. *IEEE Transactions on Knowledge and Data Engineering* 30, 2 (2017), 305–324.
[9] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, Ahmed K. Elmagarmid and Divyakant Agrawal (Eds.). ACM, Indianapolis, IN, USA, 135–146. https://doi.org/10.1145/1807167.1807184
[10] Claudio Martella, Roman Shaposhnik, Dionysios Logothetis, and Steve Harenberg. 2015. *Practical Graph Analytics with Apache Giraph* (1 ed.). Vol. 1. Apress, Berkeley, CA, Berkeley, CA. XIX, 315 pages. https://doi.org/10.1007/978-1-4842-1251-6
[11] Tahereh Pourhabibi, Kok-Leong Ong, Booi H Kam, and Yee Ling Boo. 2020. Fraud detection: A systematic literature review of graph-based anomaly detection approaches. *Decision Support Systems* 133 (2020), 113303.
[12] Yu-Xuan Qiu, Dong Wen, Lu Qin, Wentao Li, Ronghua Li, and Ying Zhang. 2022. Efficient Shortest Path Counting on Large Road Networks. *Proc. VLDB Endow.* 15, 10 (2022), 2098–2110. https://doi.org/10.14778/3547305.3547315
[13] Louise Quick, Paul Wilkinson, and David Hardcastle. 2012. Using Pregel-like Large Scale Graph Processing Frameworks for Social Network Analysis. In *International Conference on Advances in Social Networks Analysis and Mining, ASONAM 2012, Istanbul, Turkey, 26-29 August 2012*. IEEE Computer Society, Istanbul, Turkey, 457–463. https://doi.org/10.1109/ASONAM.2012.254
[14] Marko A. Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, PA, USA, October 25-30, 2015*, James Cheney and Thomas Neumann (Eds.). ACM, Pittsburgh, PA, USA, 1–10. https://doi.org/10.1145/2815072.2815073
[15] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, Farmington, PA, USA, 472–488. https://doi.org/10.1145/2517349.2522740
[16] Daniel Thilo Schroeder, Johannes Langguth, Luk Burchard, Konstantin Pogorelov, and Pedro G Lind. 2022. The connectivity network underlying the German's Twittersphere: a testbed for investigating information spreading phenomena. *Scientific reports* 12, 1 (2022), 4085.
[17] Daniel Thilo Schroeder, Kevin Styp-Rekowski, Florian Schmidt, Alexander Acker, and Odej Kao. 2019. Graph-based Feature Selection Filter Utilizing Maximal Cliques. In *Sixth International Conference on Social Networks Analysis, Management and Security, SNAMS 2019, Granada, Spain, October 22-25, 2019*, Mohammad A. Alsmirat and Yaser Jararweh (Eds.). IEEE, Granada, Spain, 297–302. https://doi.org/10.1109/SNAMS.2019.8931841
[18] Philip Stutz, Abraham Bernstein, and William W. Cohen. 2010. Signal/Collect: Graph Algorithms for the (Semantic) Web. In *The Semantic Web - ISWC 2010 - 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I (Lecture Notes in Computer Science, Vol. 6496)*, Peter F. Patel-Schneider, Yue Pan, Pascal Hitzler, Peter Mika, Lei Zhang, Jeff Z. Pan, Ian Horrocks, and Birte Glimm (Eds.). Springer, Shanghai, China, 764–780. https://doi.org/10.1007/978-3-642-17746-0_48
[19] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.
[20] Hongwei Wang, Fuzheng Zhang, Jialin Wang, Miao Zhao, Wenjie Li, Xing Xie, and Minyi Guo. 2018. RippleNet: Propagating User Preferences on the Knowledge Graph for Recommender Systems. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM 2018, Torino, Italy, October 22-26, 2018*, Alfredo Cuzzocrea, James Allan, Norman W. Paton, Divesh Srivastava, Rakesh Agrawal, Andrei Z. Broder, Mohammed J. Zaki, K. Selçuk Candan, Alexandros Labrinidis, Assaf Schuster, and Haixun Wang (Eds.). ACM, Torino, Italy, 417–426. https://doi.org/10.1145/3269206.3271739
[21] Tian Wang, Hamid Krim, and Yannis Viniotis. 2013. A generalized Markov graph model: Application to social network analysis. *IEEE Journal of Selected Topics in Signal Processing* 7, 2 (2013), 318–332.
[22] Ye Wang, Qing Wang, Henning Koehler, and Yu Lin. 2021. Query-by-Sketch: Scaling Shortest Path Graph Queries on Very Large Networks. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, Virtual Event, 1946–1958. https://doi.org/10.1145/3448016.3452826
[23] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. 2017. Tux^2: Distributed Graph Computation for Machine Learning. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, Aditya Akella and Jon Howell (Eds.). USENIX Association, Boston, MA, USA, 669–682. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/xiao