User-friendly and Efficient Distributed Graph Processing

by

Yongzhe Zhang

Dissertation

submitted to the Department of Informatics in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

The Graduate University for Advanced Studies, SOKENDAI
September 2020

Abstract

With the increasing demand of analyzing large-scale graphs generated by the modern applications, lots of research has been invested in distributed graph systems in order to process massive graphs efficiently in memory. The mainstream graph analytics systems can be classified into two classes, the vertex-centric paradigm and the linear algebra approach, each having their own advantages and disadvantages when evaluated using the following two criteria, the user-friendliness of the programming interface, and the performance and scalability on distributed-memory, both of which are important for effectively dealing with the massive graphs in practice. The vertex-centric paradigm uses relatively low-level programming interface and makes the development of distributed graph algorithms unintuitive. Even though there are high-level domain-specific languages (DSLs) proposed to ease the programming, it is still difficult for ordinary users to choose proper optimizations to ensure high efficiency. On contrary, the linear algebra approach has a concise high-level programming interface using standardized matrix and vector operations, but the optimization is much more tricky and the efficiency of this approach is yet not satisfactory for many graph algorithms.

In this thesis give a comprehensive overview of the existing graph analytics systems, analyze the difficulties in achieving both user-friendliness and efficiency, and propose our solution for achieving both goals. Our main contribution is a vertex-centric graph analytics framework using our domain-specific language (DSL) as the programming interface with a powerful back end for efficient graph analytics. Our framework is built on three key techniques, a more expressive high-level DSL to ease vertex-centric programming by hiding the message passing from users, an efficient vertex-centric back end that can arbitrarily combine various optimizations in the same vertex-program, and a novel compilation technique from our DSL to the back end as

well as the cost-based compilation technique to choose the best optimizations for a graph algorithm. The resultant framework achieved comparable performance to the state-of-the-art vertex-centric system.

We also made efforts to designing user-friendly and efficient graph analytics systems in the language of linear algebra. We currently focus on the linear algebra graph algorithms and their efficient implementation, and our results include a new connected component algorithm FastSV and Boruvka's minimum spanning forest algorithm, both of which outperform the state-of-the-art distributed algorithm significantly with our algorithm specific optimizations. In the future, we are interested in compilation techniques to make the detection of optimizations viable under the standardized linear algebra graph APIs.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Background

In the era of big data, a huge amount of data are generated every day from our daily life, like the creation of web pages in the Internet, the increase of user data in web or smartphone applications, the burst of user-generated contents on the social media, the collection of scientific data and so on. In many of these areas, graphs play a very important role in representing the data since the vertices and edges can naturally describe many real-world structures, for example the world wide web, users and their relations in a social network, and geography or biological information. Consequently, there is an increasing demand of analyzing such large-scale graph data in billion or even trillion scale, in order to extract useful information or discover valuable insights from the data.

The need of analyzing graph data in such large scale quickly introduces a big challenge to us that the graph size can easily exceed the memory capacity of a single commodity computer. Considering the extremely high cost of out-of-core data access,

lots of research has been devoted to distributed graph processing, a technique that can analyze massive graphs using the memory space of a bunch of computation devices, in order to make large-scale graph processing practical and efficient. Particularly, we assume that each device has its own physical memory and the local memory access is efficient, but all these devices are connected by a network so that accessing remote data on another computer has a significant communication cost. The main goals of distributed graph computing are *scalability* and *efficiency*: the memory requirement can always be satisfied by adding more computation devices, and a good speedup can be achieved in the meantime.

However, processing massive graphs on distributed-memory is known to be challenging, and the difficulties come from many aspects of a graph analytic task. First of all, graph algorithms running on a distributed-memory architecture are hard to design and implement. Due to the memory limitation, each computation node can only store a subgraph during the computation, and all the nodes have to communicate with each other during the graph computation to exchange necessary information for the graph computation. The combination of the computation and communication makes distributed graph algorithms very different from the sequential algorithms, and currently there is no straightforward solution to do this transition. We can see that many of the graph algorithms designed for distributed-memory are more complicated and less work-efficient. Second, graph computations are intrinsically hard to parallel on distributed-memory since it is usually difficult to divide a graph computation to small and independent tasks. A typical example is the label propagation in the form of breadth-first search (BFS), which is used as a subroutine in many distributed graph algorithms. Starting from the user-specified source vertices, or we say the initial *frontier*, the computation has to be divided to multiple runs each computing the next frontier based on the current one, and the potential challenges include the limited parallelism when the frontier is too small or load balancing problem due to some high-degree vertices becoming the bottleneck in the parallel execution. Third, when running a distributed graph algorithm on a cluster, there are lots of practical issues that affect the reliability or the performance of the graph computation, such as the risk of node failures, the low memory throughput issue due to the poor locality of the graph computation, or the load balancing issue caused by the heterogeneity in computation devices or the skewed degree distribution in practical graphs.

In the past decade, great efforts have been devoted to making large-scale graph processing practical, and lots of graph analytics systems are proposed to simplify the design and implementation of distributed graph algorithms and make the graph computation scalable and efficient on large clusters. The key is the computation model that bridges the algorithmic thinking and the parallel execution. Essentially, a computational model is an abstract machine with predefined APIs so that different graph computations can choose whatever APIs they need and reuse the parallel implementation of these APIs on distributed-memory. The computational model plays a very important role in graph processing. On one hand, it should be easy to use so that users can really find it helpful in simplifying the development of large-scale graph applications, or we say it should be user-friendly. On the other hand, the computational model decides how the graph computation is parallelized, and the users always hope the performance to be as high as possible, which in other words is the efficiency. Existing distributed graph analytics systems can be roughly categorized into two classes based on the computational model they use, which are the systems using the vertex-centric model and the systems using the linear algebra approach.

The most popular computational model is the "thinking like a vertex" paradigm (a.k.a. the vertex-centric model), which is first proposed by the Pregel [1] system and is quickly adopted by many graph analytics systems[2, 3]. The key idea is that, for many graph algorithms, we can represent them using a *vertex-program* executed on every vertex, and the whole graph computation is an iterative procedure consisting of synchronous rounds, each of which performs the vertex-centric computation and the synchronized communication phase between the vertices. Graph computation in this form is much easier to parallelize, since in each iteration there is no dependency between the computation on any two vertices, and a graph computation with massive independent tasks can achieve high parallelism. We consider the vertex-centric model a bottom-up design since a programmer first needs to understand how the system works and then tries to fit the graph algorithm into the model.

The linear algebra approach is yet another effective solution for large-scale graph analytics. By regarding the graph as a sparse adjacency matrix, many graph computations can be described as an imperative program using a small set of matrix and vector (linear algebra) operations, and the parallelization of such program has been studied on many hardware architectures including the distributed-memory systems [4]

and multicore CPU [5] or GPU systems [6]. Recently, GraphBLAS [5] defines a standard set of linear algebra operations (and C APIs [7]) for implementing graph algorithms. The linear algebra approach follows a top-down design: a programmer only considers how a graph computation is composed by the linear algebra operations, and its parallel implementation is obtained by choosing a suitable implementation on the target architecture.

1.2 Limitations

The rapid development of the graph processing frameworks in both computational models not only shows new ideas for improving large-scale graph analytics, but also reveals the limitations of the existing solutions. A crucial problem at present is that, with various sophisticated techniques proposed to improve graph analytics frameworks, it is becoming more difficult to have the user-friendliness and efficiency at the same time.

1.2.1 Vertex-centric paradigm

Pregel [1] is the most popular vertex-centric model used by the mainstream graph processing frameworks, but implementing graph algorithms in this model is known to be verbose and error-prone. In this model, a graph computation is split into synchronous rounds called *supersteps* mediated by *message passing*, and within each superstep, all the vertices execute the same user-defined *compute()* function in parallel, which can read the messages sent to it in the previous superstep, modify its own state, and send messages to other vertices. Global barrier synchronization happens at the end of each superstep, delivering messages to their designated receivers before the next superstep. This model is initially proposed to solve some rather simple algorithms like the single-source shortest path (SSSP) and PageRank, but when the algorithm become complicated, for example consisting of multiple stages and complicated data dependencies [8], users need to write an exceedingly complicated *compute()* function as the loop body to encodes all the stages of the algorithm.

Pregel [1] stands for a particular vertex-centric system based on the Bulk-synchronous parallel (BSP) model [9] and inter-vertex messages passing, but some frameworks [10,

11, 12, 13, 14, 15] propose alternative models that hide the message passing from users. For example, in PowerLyra [12] a vertex-program by default reads the state of the neighboring vertices in every iteration and combine their values into a single message value, and the system is designed to reduce the communication cost for power-law graphs by using more balanced partitioning strategy. Although using these Pregel variations can indeed simplify the programming and even improve the efficiency in some scenarios, they also pose a restriction that a vertex can only communicate with neighboring vertices, which is insufficient to implement some useful distributed graph algorithms like the connected component algorithm [16, 17] and the minimum spanning tree algorithm [18]. We consider it a significant drawback compared to the original Pregel system. Some other attempts have been made to ease Pregel programming by proposing domain-specific languages (DSLs), such as Green-Marl [19] and Fregel [20], but the main issue is similar to the aforementioned systems with a simplified computation model that not all commonly used graph algorithms can be implemented. Essentially, these DSLs do not support general *remote data access*, reading or writing attributes of other vertices through references, which makes them less expressive in practice.

In addition to the programming interface, there are actually many potential issues in Pregel's model that may hurt the performance, such as imbalanced workload (a.k.a. skewed degree distribution) [11, 21, 8], redundancies in communication [22, 23, 21] and low convergence speed [24, 25, 26], and lots of research has shown that it is important to introduce optimizations for dealing with these problems. However, there remains one challenge: although the usefulness of these optimizations are well demonstrated in solving simple algorithms such as PageRank and single-source shortest path (SSSP)[1], it is hard to combine them together to implement complex algorithms, where we may have to deal with multiple performance issues at the same time. A typical example is the Shiloach-Vishkin [16] algorithm for finding connected components which suffers two different issues in different stage of the computation (see Section 2.1.3 for more details). Unfortunately, all the existing vertex-centric graph systems lack a modular design and it is yet impossible for users to combine the optimizations in the same program.

[1]PageRank and SSSP are basically a loop executing a simple computation kernel.

1.2.2 Linear algebra approach

The linear algebra approach is yet another successful programming model for large-scale graph processing, which is considered to be more elegant than to the vertex-centric paradigm. In particular, by using the standardized programming interface called GraphBLAS [5], users can easily transform a graph computation from other platforms to distributed-memory by simply using a GraphBLAS compliant library. However, the linear algebra approach also has limitations in its programming interface and efficiency. Generally speaking, a linear algebra program is typically hard to optimize since the matrix and vector operations are too generic that lacks the necessary information for optimization.

To clearly see this problem, let us take the matrix-vector multiplication in linear algebra as an example, which plays the central role in many graph computations. The efficiency of this operation in fact depends on many things such as the computation platform, the sparsity of the graph, the data structure used in the program, the size of the vector and so on, and there is no such an implementation that generally performs well in every situation. For this simple operation, GraphBLAS actually provides two versions `mxv` and `vxm` that are similar in functionality but different in the order of parameters (`vxm` is the vector-matrix multiplication). Users can choose either of them in their program, but the performance of `mxv` and `vxm` are not guaranteed to have the same performance, and in the library SuiteSparse:GraphBLAS [27] they are implemented in different ways to handle the sparse and dense vectors respectively, while the misuse of these two functions may suffer a performance penalty. Therefore, it is not easy for an ordinary user to implement an efficient graph algorithm without knowing the details of the implementation of GraphAPIs in the library.

In addition to the redundancy in the GraphBLAS API standard, tuning a graph computation in linear algebra can be also very tricky, and the true difficulty is also in its programming interface. There are lots of research focusing on the performance issues in the vertex-centric paradigm, and many of them can be solved transparently since the system designer can easily capture the necessary information (e.g. the number of active vertices in a superstep or the degree distribution of the vertices) and optimize the computation as a whole. However, a graph computation in linear algebra consists of atomic operations, which at least causes to problems. First, the linear algebra operation

cannot capture the statistical information of the whole graph. For example, it is not possible to know the number of active vertices during the computation, because we have no idea which vector stores the activation information. Second, the cause of the performance issue in linear algebra is often the combination of linear algebra operations, and it cannot be solved by optimizing individual APIs. In practice, the most popular linear algebra library CombBLAS [4] for distributed-memory is not GraphBLAS compliant in order to enable application-aware APIs to optimize its performance.

1.3 Contribution

In this thesis, we basically answer the following question: how can we build a graph analytics system that is both user-friendly and highly efficient. Up to now, there are already lots of graph frameworks proposed in either the vertex-centric model or the linear algebra approach, but we believe that none of them is satisfactory. This work reviews many of the techniques to improve the user-friendliness and efficiency of graph analytics system and proposes our unique solution to fulfill both goals. In our solution, the user's program for graph analytics is written in a high-level domain-specific language that is easy to understand, and it is the compiler's duty to transform the program to a graph analytics system's low-level APIs. Furthermore, by using program analysis techniques, the compiler can choose the best combination of optimizations that are feasible on that graph system to maximize the performance. Although our current results are mainly obtained in the vertex-centric paradigm, we give the first practical solution that achieves user-friendliness and high efficiency at the same time, and we also show promising results in the linear algebra approach.

The technical contributions of this thesis are summarized as follows. For the vertex-centric paradigm, the main contribution is the programming interface to make it easy to use:

- First, we design and implement Palgol, a powerful DSL that supports both remote reads and writes, and allows programmers to use a more declarative syntax called *chain access* to directly read data on remote vertices. This language is based on our new high-level model for vertex-centric computation, where the

concept of *algorithmic supersteps* is introduced as the basic computation unit for constructing vertex-centric computation in such a way that remote reads and writes are ordered in a safe way. We demonstrate the power of Palgol by working on a set of representative examples, and the efficiency of Palgol is comparable with hand-written Pregel code for many representative graph algorithms on practical big graphs.

- Second, we provide Pregel with a channel-based programming interface, which is a natural extension of Pregel's monolithic message mechanism and allows users to add new optimizations in the system in a modular way. To demonstrate the power of the channel interface, we implement three optimizations as special channels and show how they are easily composed to optimize complex algorithms such as the SV algorithm. We fully implement the system and the experiment results convincingly show the high efficiency of our system. The channel interface itself reduces the communication cost for complex algorithms, and the three optimized channels improves the runtime by 3.50% for PageRank, 4.41% for pointer jumping and 5.20% for label propagation. Specially, the composition of different optimizations makes the SV algorithm 3.39% faster than the best implementation available now.

- Third, to transform Palgol to the efficient but more complex system Pregel-channel, we propose a core language SQL-core to describe various graph computations. By using an intuitive tabular representation for graphs, we show that a wide range of scalable graph computations can be represented as the inner join of a series of tables followed by the filtering and aggregation. We show how to obtain optimal vertex-centric programs by demonstrating that two useful optimizations [21, 28] can be easily detected in our high-level model, and the integration of such analysis can be achieved by a simple extension of our join-based compilation algorithm. We have fully implemented the compiler, and the experiment results convincingly show that our compilation algorithm achieves similar efficiency for many representative graph algorithms on large graph dataset.

For the linear algebra approach, we currently focus on the efficient implementation of various graph applications. In this thesis, two graph algorithms are newly proposed in

the language of linear algebra and their efficient implementation on distributed-memory are studied. In detail,

- We develop a simple and efficient algorithm FastSV for finding connected components in distributed memory, which is based on the Shiloach-Vishkin algorithm but uses novel hooking strategies for fast convergence. We present FastSV using a handful of GraphBLAS operations and implement the algorithm in CombBLAS for distributed-memory platforms. We dynamically use sparse operations to avoid redundant work and optimize MPI communication to avoid bottlenecks. Both shared- and distributed-memory implementations of FastSV are significantly faster than the state-of-the-art algorithm LACC. The distributed-memory implementation of FastSV can find CCs in a hyperlink graph with 3.27B vertices and 124.9B edges in just 30 seconds using 262K cores of a XC40 supercomputer.

- We also present a linear algebra formulation for Boruvka's MST algorithm and provide a message-efficient parallelization on distributed-memory. We conduct experiments to show that our parallelization of Boruvka's algorithm achieves much higher performance and outperforms the state-of-the-art MSF problems significantly. We are also the first to use Boruvka's algorithm for solving the CC problem on distributed-memory, and we show that it is even better than the fastest linear algebraic FastSV [17] algorithm on both convergence and performance.

The rest of the thesis is organized as follows. Chapter 2 is a review of the previous works in the field of distributed graph processing. Chapter 3 discusses the vertex-centric graph processing systems and the channel mechanism we designed to integrate various optimizations in a compositional way. Chapter 4 presents our design of two domain-specific languages (DSLs), the Palgol language to ease the vertex-centric programming and the SQL-core language with its novel compilation technique to find opportunities for deploying optimizations using our channel-based Pregel system. Chapter 5 reviews linear algebra solution for large-scale graph processing, in which we propose two new graph algorithms for solving the connected component problem and the minimum spanning forest problem. Chapter 6 conclude this thesis and discuss the future work.

2

Distributed Graph Processing: A Review

In this chapter, we review the two mainstream distributed graph processing system and discuss their limitations in efficiency and productivity.

2.1 "Thinking like a vertex" paradigm

Graph processing on distributed-memory is completely different from the conventional graph processing on a single-core or multi-core machine that has a decentralized design pattern. Conventionally, a graph algorithm designed for a single machine assumes that the entire in graph is randomly accessible in memory, and a centralized computational agent processes the graph in a sequential, top-down manner [2]. However for large-scale graphs occupying terabytes or more, each machine holds only a portion of the data and none of them can see the entire graph. This restriction poses a big challenge to algorithm designer, because some of the standard graph operations (e.g., depth-first-search) may no longer work due to the high latency of accessing data on a remote node.

The vertex-centric frameworks are platforms that iteratively executes a user-define *vertex-program* on every vertex of the graph, which can manipulate the current vertex's state and exchange data with the other vertices. The vertex-program is executed across vertices of the graph synchronously or may also be executed asynchronously, and usually the computation terminates after a fixed number of iterations or all vertices have converged. The vertex-centric programming model is less expressive but it can easily achieves high parallelism or scalability since in any iteration the computation tasks assigned to all the vertices are independent.

2.1.1 The Pregel system

Google's Pregel [1] is one of the most popular frameworks for processing large-scale graphs. The Pregel system takes a directed graph as input (an undirected graph can be treated as a directed graph that always has edges in both directions), where user-specified values can be associated on either vertices or edges. A Pregel computation follows the bulk-synchronous parallel (BSP) model [9] and it consists of a series of supersteps separated by global synchronization points. In each superstep, the vertices compute in parallel executing the same user-defined function (usually the `compute()` method) that expresses the logic of a given algorithm. A vertex can read the messages sent to it in the previous superstep, mutate its state, and send messages to its neighbors or any known vertex in the graph. The termination of the algorithm is based on every vertex voting to halt. Each vertex is associated with a flag indicating whether it is active, and initially it is set to true. During the computation, a vertex can deactivate itself by invoking a `vote_to_halt()` method, and it can be reactivated externally by receiving messages

Pregel provides the message passing interface for vertex-to-vertex communication and aggregator for global communication.

Message passing and the combiner. In Pregel, vertices communicate directly with each other by sending messages, where each message consists of a message value and a destination. The combiner optimization [1] is applicable if the receiver only needs the aggregated result (like the sum, or the minimum) of all message values, in which case the system is provided an associative binary function to combine messages for the same destination whenever possible.

Aggregator. Aggregator is a useful interface for global communication, where each active vertex provides a value, and the system aggregates them to a final result using a user-specified operation and makes it available to all vertices in the next superstep.

2.1.2 Domain-specific languages and their problems

Despite the power of Pregel, it is a big challenge to implement a graph algorithm correctly and efficiently [21], especially when the algorithm consists of multiple stages and complicated data dependencies. For such algorithms, programmers need to write an exceedingly complicated *compute()* function as the loop body, which encodes all the stages of the algorithm. Message passing makes the code even harder to maintain, because one has to trace where the messages are from and what information they carry in each superstep.

Some attempts have been made to ease Pregel programming by proposing domain-specific languages (DSLs), such as Green-Marl [19] and Fregel [20]. These DSLs allow programmers to write a program in a compositional way to avoid writing a complicated loop body, and provide more convenient communication primitives to avoid explicit message passing, e.g. fetching a particular attribute from all the neighbors of a vertex. The compiler then transforms the communication primitives to obtain a Pregel code using the message passing interface. However, there are still several crucial problems that are yet not addressed by these DSLs.

Limited expressiveness.

In all of the existing DSLs, there is a severe restriction on data access that each vertex can only access data on their neighboring vertices. In other words, they do not support general *remote data access* — reading or writing attributes of other vertices through references.

Remote data access is, however, important for describing a class of Pregel algorithms that aim to accelerate information propagation (which is a crucial issue in handling graphs with large diameters [21]) by maintaining a dynamic internal structure for communication. For instance, a parallel pointer jumping algorithm maintains a tree (or list) structure in a distributed manner by letting each vertex store a reference to its current parent (or predecessor), and during the computation, every vertex constantly

exchanges data with the current parent (or predecessor) and modifies the reference to reach the root vertex (or the head of the list). Such computational patterns can be found in algorithms like the Shiloach-Vishkin (SV) connected component algorithm [21], the list ranking algorithm (see Section 4.1.3) and Chung and Condon's minimum spanning forest (MSF) algorithm [18]. However, these computational patterns cannot be implemented with only neighboring access, and therefore cannot be expressed in any of the existing high-level DSLs.

Lack of a global view.

The vertex-centric paradigm requires the users to explicitly describe data communication between the vertices, which is difficult when dealing with communication patterns that exceed a vertex's direct neighbors. As an example, let us look at the triangle counting problem, which reflects a general problem in many graph algorithms. In a directed graph, the triangle counting can be solved by counting the number of distinct paths in the pattern of $u \to v \to w \to u$, a path from vertex u to itself through v and w. The most straightforward way is to enumerate all possible u, v and w that are connected by the edges, where v is chosen from u's adjacent list and w is chosen from v's. When doing so in the vertex-centric model, we are immediately faced with the following questions:

- on which vertex's perspective should we implement the `compute()` function;
- what data is required on each vertex; and
- what is the communication cost.

These questions do not always have an obvious answer, and it requires users to carefully consider all the possibilities, which is in general a hard task. Existing DSLs are unfortunately not helpful to this problem, since their high-level models lack a global view of data communication.

No guidance for optimization.

Pregel's message passing mechanism can easily cause performance issues in communication. For example, in PageRank's Pregel implementation [1], every vertex needs to send its tentative ranking value to all the neighbors in each iteration. Such communication pattern has to be implemented by a vertex sending individual messages to neighbors containing the same value, which causes a redundancy problem. There are

lots of research [23, 22, 29, 21] showing that exploring such high-level communication patterns can lead to powerful optimizations that reduce the communication cost significantly. However, Pregel’s model does not give any hint on when and how the users can optimize their programs. This issue can be potentially solved by program analysis using a high-level DSL, but existing DSLs are not expressive enough for supporting such analysis, and yet there is no such vertex-centric graph analytics system that can support various optimizations at the same time.

2.1.3 Problem of Pregel’s monolithic message interface

Pregel is designed to support iterative computations for graphs, and it is indeed suitable for algorithms like the PageRank or SSSP. However, it is noteworthy that vertex-centric graph algorithms are in general complex. Even for some fundamental problems like connected component (CC), strongly connected component (SCC) and minimum spanning forest (MSF), their efficient vertex-centric solutions require multiple computation phases, each having different communication patterns [21, 30]. For such complex algorithms, all the computation phases have to share Pregel’s message passing interface, which causes the following problems:

- When different message types are needed in different computation phases, the Pregel’s message interface has to be instantiated with a type that is large enough to carry all those message values.
- Usually, we can no longer optimize any of the communication patterns in these computation phases, since the system cannot distinguish which message is to be optimized.

As mentioned before, these are the consequences of Pregel’s monolithic message mechanism, which may not only increase the message size, but also prevent the possible optimizations to be applied.

2.1.4 Vertex-centric systems other than Pregel

The vertex-centric graph analytics systems are not limited to the Pregel-like systems and several graph processing frameworks propose different vertex-centric programming

model to deal with various issues in the graph computation. In this part, we briefly summarize their differences with Pregel and the problems.

However, the majority of them (e.g., [10, 11, 12, 13, 14, 15]) are functionally similar to the sparse-matrix vector multiplication, while Pregel's model is yet the only model that can deal with both edge removal and non-neighborhood communication (analogous to the `select`, `assign` and `extract` operations in linear algebra), which are necessary for Boruvka's algorithm and some other scalable graph computations [17, 31, 32]. We should note that none of the existing linear algebra graph libraries [4, 33, 13] have fully implemented these operations, and our work is the first linear algebra parallelization of Boruvka's algorithm on distributed-memory.

2.2 The linear algebra approach

The idea of using linear algebra to implement graph algorithms is based on the duality between the fundamental operations on graphs/matrices – BFS and matrix multiplication. The graph is regarded as a sparse adjacency matrix where each element at the position (i, j) is an edge connecting the vertices i and j (the vertices are indexed from $0..n-1$). Then, the generalized matrix-vector multiplication can be considered as a single-step label propagation where each vertex in the vector propagates its state (the value in the vector on the corresponding position) to all the neighbors, and the received values on every vertex are combined together and written to the output vector. The label propagation is the most useful in many graph algorithms like the single-source shortest path (SSSP), PageRank, connected component (CC) and so on. In addition to the matrix-vector multiplication, we summarize the most useful linear algebra operations in Figure 2.1

2.2.1 Merits of the linear algebra approach

Expressing graph algorithm in the language of linear algebra is attractive, and the main advantages of the linear algebra approach is summarized as follows:

- **conciseness:** By treating the graph as a sparse matrix, the core operations in many scalable graph algorithms can be presented by a small set of linear algebra operations, and these operations are parallelized by performance experts on

operation	approximate MATLAB analog
matrix multiplication	`C=A*B`
element-wise operations	`C=A+B` and `C=A.*B`
reduction to a vector or scalar	`s=sum(A)`
apply unary operator	`C=-A`
transpose	`C=A'`
submatrix extraction	`C=A(I,J)`
submatrix assignment	`C(I,J)=A`

Figure 2.1: A brief summary of linear algebra APIs.

various architectures. In this sense, the linear algebra approach is particularly preferable for distributed graph processing since it makes the communication completely transparent to the programmers.

- **scalability:** The implementation of matrix and vector operations on distributed-memory can make use of both MPI and OpenMP to maximize its scalability, and a recent work [31] using the Combinatorial BLAS [4] library successfully solves the connected component problem on a graph with more than 50B edges, using a Cray XC40 supercomputer with 4K nodes (262K cores).

- **portability:** Recently, GraphBLAS [5] defines a standard set of linear-algebraic operations for implementing graph algorithms, which makes it much easier for programmers to transform a linear algebra graph computation from one platform to another.

3

A custom Pregel system with optimizations as plug-ins

Pregel's vertex-centric paradigm is widely adopted in many graph analytics systems, but as mentioned in the previous chapter, this model suffers performance issues when dealing with graph algorithms with complex communication patterns (e.g. the connected component algorithm and the minimum spanning forest algorithm) since Pregel's monolithic message mechanism cannot make use of multiple optimizations at the same time.

In this chapter, we propose a new approach to composing various optimizations together, by making use of the interface called *channel* [34] as a replacement of Pregel's message passing mechanism. Informally, a channel is responsible for sending or receiving messages of a certain pattern for some purpose (such as reading all neighbors' states, requesting data from some other vertex and so on). And by slicing the messages by their purpose and organizing them in channels, we can characterize each channel by high-level communication patterns, identify the redundancies or potential performance

Table 3.1: The APIs for standard channels.

Message-Passing Channels		Aggregator Channel
`DirectMessage(Worker<VertexT> *w);`	`CombinedMessage(Worker<VertexT> *w, Combiner<ValT> c);`	`Aggregator(Worker<VertexT> *w, Combiner<ValT> c);`
`void send_message(KeyT dst, ValT m);` `MsgIterator<KeyT, ValT> &get_iterator();`	`void send_message(KeyT dst, ValT m);` `const ValT &get_message();`	`void add(ValT v);` `const ValT &result();`

Table 3.2: The APIs for special channels targeting specific communication patterns.

Scatter-Combine	Request-Respond	Propagation (Simplified)
`ScatterCombine(Worker<VertexT> *w, Combiner<ValT> c);`	`RequestRespond(Worker<VertexT> *w, function<RespT(VertexT)> f);`	`Propagation(Worker<VertexT> *w, Combiner<ValT> c);`
`void add_edge(KeyT dst);`		`void add_edge(KeyT dst);`
`void set_message(ValT m);` `const ValT &get_message();`	`void add_request(KeyT dst);` `const RespT &get_response();`	`void set_value(ValT m);` `const ValT &get_value();`

issues, and then provide separate implementations to deal with their own issues.

The technical contributions of this work can be summarized as follows.

- First, we provide Pregel with a channel-based vertex-centric programming interface, which is intuitive in the sense that it is just a natural extension of Pregel's monolithic message mechanism. To demonstrate the power of the channel interface, we implement three optimizations as special channels and show how they are easily composed to optimize complex algorithms such as the above SV algorithm.

- Second, we have fully implemented the system and the experiment results convincingly show the usefulness of our approach. The channel interface itself contributes to an up to 76% reduction of message size especially for complex algorithms, and the three optimized channels further improve the performance of the algorithms they are applicable (3.50x for PageRank, 4.41x for Pointer-Jumping and 5.20x for weakly connected components). Specially, the composition of different optimizations makes the SV algorithm 3.39x faster than the best implementation available now.

3.1 Programming with channels

The channel mechanism is designed to help users organize the communications in vertex-centric graph algorithms. Concretely speaking, the channels are message

containers equipped with a set of methods for sending/receiving messages or supporting a specific communication pattern (see Table 3.1 and Table 3.2 for the standard and optimized channels; the details are in Section 3.2). In this section, we first introduce the programming interface using the PageRank example, then we show how different optimizations can be easily composed via channels in a more complex algorithm called the SV [16].

3.1.1 A standard PageRank implementation using channels

Writing a vertex-centric algorithm in our system using the standard channels is rather straightforward for a Pregel programmer. We present a PageRank Implementation in Figure 3.1, which is basically obtained from a Pregel program (a vertex-centric `compute()` function with a parameter of received messages from the previous superstep) by replacing the sending/reading of messages by one or more user-defined message channel's send/receive methods.

In the first 30 supersteps, each vertex sends along outgoing edges (if exists) its tentative PageRank divided by the number of outgoing edges (lines 21–25), over a user-defined message channel `nbr`. This channel is an instance of `CombinedMessage`, which requires a combiner to be provided in its constructor (line 9). In the next superstep, every vertex gets the sum of the message values arriving on this channel (lines 18) and calculates a new PageRank. To avoid PageRank lost in dead ends (vertices without outgoing edges), we need a *sink* node to collect the PageRank from those dead ends and redistribute it to all nodes, which is implemented by an aggregator `agg` using the addition operator (line 9). Then, in line 27, users explicitly add the PageRank of the dead ends to the aggregator, and in the next superstep the sum is returned by the aggregator's `result()` method (line 16).

All the computation logic and the channels are written in a user-defined class called `PageRankWorker` that inherits from the `Worker` class in our system. The type of vertex ID and value type are packed in to the `VertexT` type and provided to the `Worker` class. We leave the explanation of `Worker` in the next section, and users just keep in mind that programs in our system are constructed in this way.

```
1  using VertexT = Vertex<int, PRValue>;
2  auto c = make_combiner(c_sum, 0.0); // a combiner
3  class PageRankWorker : public Worker<VertexT> {
4  private:
5     // two channels are defined here
6     CombinedMessage<VertexT, double> nbr;
7     Aggregator<VertexT, double> agg;
8  public:
9     PageRankWorker():nbr(this, c), agg(this, c) {}
10
11    void compute(VertexT &v) override {
12       if (step_num() == 1) {
13          value().PageRank = 1.0 / get_vnum();
14       } else {
15          // s: the pagerank of the "sink node"
16          double s = agg.result() / get_vnum();
17          value().PageRank = 0.15 / get_vnum()
18                + 0.85 * (nbr.get_message() + s);
19       }
20       if (step_num() < 31) {
21          int numEdges = value().Edges.size();
22          if (numEdges > 0) {
23             double msg = value().PageRank / numEdges;
24             for (int e : value().Edges)
25                nbr.send_message(e, msg);
26          } else
27             agg.add(value().PageRank);
28       } else
29          vote_to_halt();
30    }
31 };
```

Figure 3.1: PageRank implementation using channels.

3.1.2 Channels and optimizations

In our channel-based system, we offer a set of optimizations as special channels (in Table 3.2), which can be regarded as more efficient implementations (compared to the standard message passing channels) of several communication patterns. Here, we demonstrate how to enable the scatter-combine optimization (which deals with the "static messaging pattern") for PageRank. The details of this optimization will be presented in Section 3.2.

Given a channel-based PageRank implementation in Figure 3.1, what we need to do is simply switching the standard message channel `msg` to the scatter-combine channel. First, we change the definition of `nbr` as follows:

```
5     // change to the scatter-combine channel
6     ScatterCombine<VertexT, double> nbr;
```

Then, in the `compute()` function, we initialize the scatter-combine channel (by

invoking the `add_edge()` method) before actually sending any data, which is done in the first superstep as below:

```
12    if (step_num() == 1) {
13       value().PageRank = 1.0 / get_vnum();
14       // provide the graph topology to the channel
15       for (int e : value().Edges)
16          nbr.add_edge(e);
17    } ...
```

Finally, we switch to the scatter-channel's dedicated interface for message passing, which is `set_message()` indicating a unique message value for all neighbors:

```
22    if (numEdges > 0) {
23       double msg = value().PageRank / numEdges;
24       // no need to specify the destination
25       nbr.set_message(msg);
26    } ...
```

The rest of the program remains the same. Our experiments (Section 3.3.2) show that, by switching to the scatter-combine channel, the PageRank immediately gets 3x faster, and all the programmer need to understand is the high-level abstraction of each channel.

3.1.3 Composition of channels

In this part, we use a more complicated example called the Shiloach-Vishkin (SV) algorithm [16] to show that, users can easily combine different optimizations (channels) to handle multiple performance issues in the same program.

The SV Algorithm

The SV algorithm is in general an adaptation of the classic union-find algorithm [35] to the distributed setting, which finds the connected components in undirected graphs with n vertices in $O(\log n)$ supersteps. In the SV algorithm, the connectivity information is maintained by a distributed tree structure called disjoint-set [35], where each vertex holds a pointer which points to either some other vertex in the same connected component or to itself. We henceforth use $D[u]$ to represent this pointer for vertex u. Following is the high-level description of the SV algorithm using a

domain-specific language called Palgol [36], and it compiles to Pregel+ code[1].

```
1  // initially suppose we have D[u] = u for every u
2  do
3    // enter vertex-centric mode
4    for u in V
5      // whether u's parent is a root vertex
6      if (D[D[u]] == D[u])
7        // iterate over neighbors (D[e]: neighbor's pointer)
8        let t = minimum [ D[e] | e <- Nbr[u] ]
9        if (t < D[u])
10         // modify the D field of u's parent D[u]
11         remote D[D[u]] <?= t
12     else
13       // the pointer jumping (path compression)
14       D[u] := D[D[u]]
15   end
16 until fix[D] // until D stabilizes for every u
```

Starting from n root nodes, the SV algorithm iteratively merges the trees together if crossing edges are detected. In a vertex-centric way, every vertex u simultaneously performs one of the following operations depending on whether its parent $D[u]$ is a root vertex:

- **Tree merging (lines 7–11).** If $D[u]$ is a root vertex, u sends the smallest one of its neighbors' pointer (to which we give a name t) to the root $D[u]$ and later the root points to the minimum t it receives (to guarantee the correctness of the algorithm).

- **Pointer jumping (line 14).** If $D[u]$ is not a root vertex, u modifies its pointer to its "grandfather" ($D[u]$'s current pointer). Since all the vertices below the children of root perform this operation simultaneously, it halves the distance to the current root.

The algorithm terminates when all vertices' pointers do not change after an iteration. Readers interested in the correctness of this algorithm can be found in the original paper [21] for more details.

[1]The Palgol code is presented here for easy understanding. It currently compiles to Pregel+ using only the message interface, and the the performance is close to the hand-written code [36].

Choices of Channels

In the SV algorithm, three major performance issues are identified below by analyzing the communication patterns in the algorithm.

- The load balance issue in testing whether $D[u]$ is a root vertex or not for every u. The standard implementation is to let each u send a request to its current parent $D[u]$, then the reply message is the parent's pointer. Due to the pointer jumping, the height of the tree will decrease and the width of the tree will increase, causing a few vertices with very large degree to slow down the reply phase.
- The heavy neighborhood communication in calculating the minimum parent ID of the neighboring vertices, where all vertices need to send a unique message value to all neighbors, regardless of the vertices' local state.
- The congestion issue in the modification of parent's pointer, due to the existence of high-degree vertices.

We provide the solutions to all of these issues in our system as special channels, and users just need to choose the proper channels and combine them together in the program. For the SV algorithm, the load balance issue can be avoided by the request-respond channel, the heavy neighborhood communication is optimized by the scatter-combine channel, and a message channel with combiner solves the congestion issue.

3.2 Channel implementation

In this section, we present the design of our channel mechanism and demonstrate how three interesting channels are implemented for dealing with different performance issues.

3.2.1 Overview

Figure 3.2 shows the architecture of our channel-based system. Worker is the basic computing unit in our system. When launching a graph processing task, multiple instances of workers are created, each holding a disjoint portion of the graph (a subset

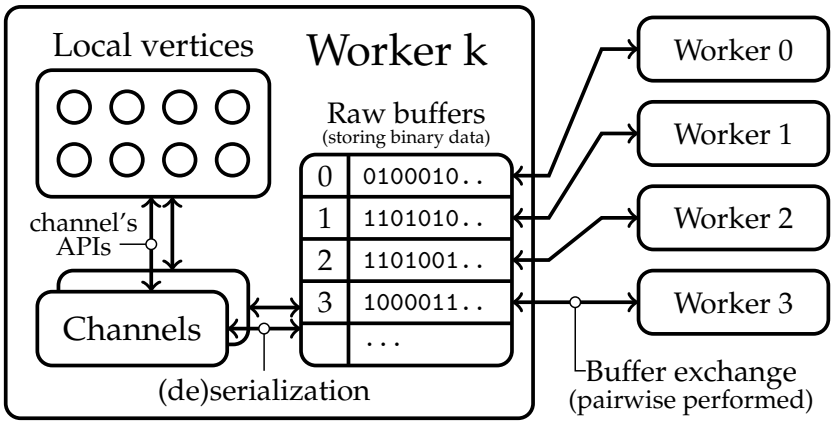


Figure 3.2: The architecture of our channel-based system.

```
1  class Channel {
2  public:
3     // initialization function
4     virtual void initialize() {};
5     // paired (de)serialization functions
6     virtual void serialize(Buffer &buff) = 0;
7     virtual void deserialize(Buffer &buff) = 0;
8     // return true for additional buffer exchange
9     virtual bool again() { return false; };
10 };
```

Figure 3.3: The core functions of the base class `Channel`.

of vertices along with their states and adjacent lists). Workers share no memory but can communicate with each other. Such big picture is common in all Pregel systems, but ours has a unique hierarchy of the components inside the worker.

In our system, channels form an independent layer inside the worker between the vertices and the raw buffers. Each worker has $M - 1$ buffers (where M is the number of workers launched by the user) for storing binary message data for each other worker, then the channels can read or write its own address space on these buffers. The system simply exchanges the contents of the buffers pairwise using the MPI in every superstep. Each channel independently implements a communication pattern (like messages passing or aggregator) and exposes its own interfaces (like `send_message(dst, msg)` `for a message channel`) to the vertex. To implement an algorithm, users should inherit the `Worker` class, override the `compute()` function and allocate the channels that are suitable for the algorithm according to the communication patterns it has.

3.2.2 Design principle

The channel mechanism mainly targets (but not limited to) a class of optimizations that handles the redundancies in communication. For example, the standard combiner optimization [1] allows the worker to combine the messages sent to the same destination by a user-defined binary operator, and the request-respond paradigm [21] merges the requests to the same destination to avoid sending redundant copies of the same value. In these optimizations, typically, each worker processes all the messages in batch and sends a more compact but equally informative message list. After the messages are delivered, the receiver worker may further process the data and dispatch the messages to the vertices.

Having such common pattern in these communication related optimizations, our channel mechanism tries to organize them in a modular way and make them work perfectly with the Pregel abstraction. Essentially, each channel is a user-specified message handler that is invoked by the worker in every superstep. The vertices actually put messages into (or fetch messages from) the channels' local storage through each channel's dedicated APIs, and the message handler can access all the local data of the channel as well as current worker's states to implement a particular communication pattern. Channels are registered on the worker, so the composition of channels is actually trivial, which is accomplished by the worker carefully separating the messages of each channel in its message buffer.

3.2.3 The channel interface

Figure 3.3 shows the base class `Channel` and its core functions: `initialize()`, `serialize()` for writing data to worker's raw buffer, `deserialize()` for reading data from worker's raw buffer (after the buffer exchange) and `again()` for supporting multiple rounds of communication. All the channels in our paper are implemented as derived classes of `Channel` with proper implementations of these four functions (in particular `serialize()` and `deserialize()`).

To clearly see how the workers and channels cooperate with each other, we present the computation logic of the worker in Figure 3.4. The worker's computation is organized as synchronized supersteps. In each superstep, the worker first calls the `compute()` on every vertex, then it performs several rounds of buffer exchanges. In

```
load_graph()
foreach channel c do c.initialize()
foreach vertex v do v.set_active(true)
while (active vertex exists) // a superstep
   foreach active vertex v do this.compute(v)
   foreach channel c do c.set_active(true)
   while (active channel exists)
      foreach active channel c do c.serialize()
      buffer_exchange()
      foreach active channel c do
         c.deserialize()
         c.set_active(c.again())
      end_for
   end_while
end_while
dump_graph()
```

Figure 3.4: The computation logic of the worker for illustrating the channel mechanism.

each round, the system invokes the active channels' `serialize()` and `deserialize()` methods to exchange the data between the channels and the buffers. All channels are set to active at the beginning, but they can deactivate themselves by returning `false` in the `again()` function. Channels' `initialize()` is invoked at the beginning of the computation, in which the channel can access the basic information of the graph (like graph size, number of vertices on the current worker) for initialization. While not explicitly presented in the code, the channels can activate vertices through the `Worker`'s interface by providing the vertex's ID or local index. That is how our system simulates the voting-to-halt mechanism of Pregel.

3.2.4 Case studies

As the last part of this section, we demonstrate how to implement three optimizations, which target three important performance issues in vertex-centric graph processing.

Scatter-Combine Channel

The scatter-combine abstraction is a common high-level pattern appeared in many single-phase algorithms such as PageRank, single-source shortest path (SSSP) and connected component (CC). The communication in this model is captured by a `scatter()` function on each vertex to send a unique value to all neighbors, and a `combine()` function to combine the messages for each receiver. We focus on a special

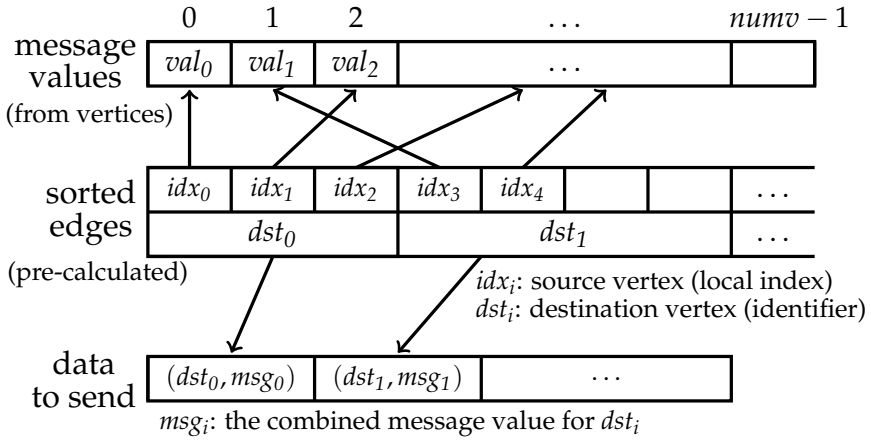


Figure 3.5: The execution logic for the scatter-combine channel.

case where *every* vertex needs to send a value to all of its neighbors[2] regardless of its local state. An iterative algorithm having such static messaging pattern will waste time repeating the same message dispatching procedure, while a proper preprocessing can greatly reduce the computation time as well as the message size.

Figure 3.5 demonstrates the computation logic of the scatter-combine channel. Suppose the vertices on an worker is indexed by `0..numv-1`, then each local edge is a pair (idx, dst) where idx refers to a local vertex and the dst can be an arbitrary vertex in the graph. We sort the edges by dst in advance, then by scanning the array of the sorted edge list once, we can quickly calculate for each destination a combined message value. This is much cheaper than the normal message routine which typically requires hashing or sorting.

The APIs for the scatter-combine channel are presented in the first column of Table 3.2. Users need to initialize the channel by adding the outgoing edges of each vertex through the `add_edge()` function before the first message sending occurs in the execution. Then, every vertex emits an initial messages using the `send_message()` interface and the combined messages for each vertex can be obtained by the `get_message()` method in the next superstep.

[2]In some algorithms like SSSP or WCC, only active vertices need to send messages, which is not the case we are targeting here.

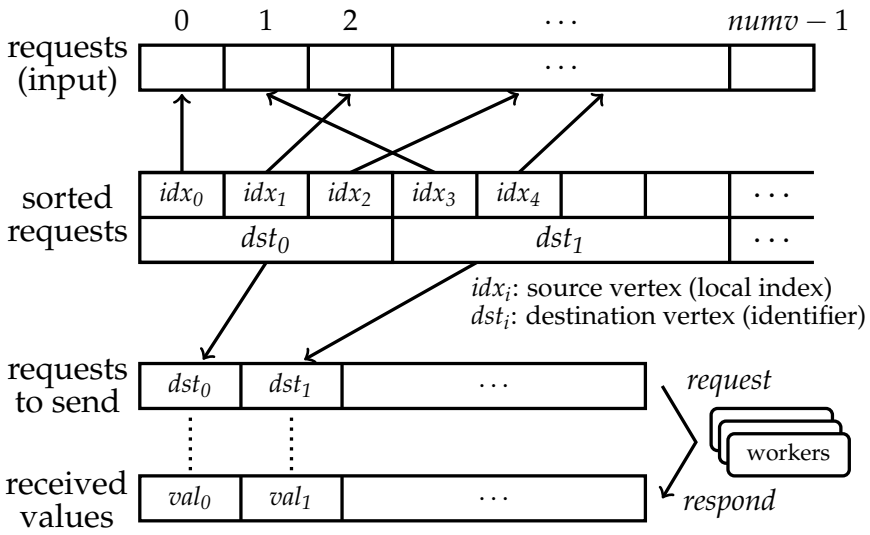


Figure 3.6: The execution logic for the request-respond channel.

Request-Respond Channel

This is a communication pattern where two rounds of message passing (say the request phase and respond phase) together form a conversation to let every vertex request an attribute of another vertex. Typically, such computation contains vertices with high degree which causes imbalanced workload in the respond phase, and the solution is to merge the requests of the same destination on each worker. More details can be found in the original paper [21].

Our implementation of this optimization is illustrated in Figure 3.6. A request is a pair (idx, dst) where idx refers to a local requester and the dst can be an arbitrary vertex in the graph. The worker sorts the requests by dst and sends exactly one message containing the worker ID to each of the unique destinations. When receiving the response values, the worker performs another scan to the sorted requests, which is sufficient to reply to all the requesters.

The middle column of Table 3.2 shows the APIs of the request-respond channel. When creating the channel, users need to provide a function that generates a response value from a vertex's state. The whole procedure is implemented in an implicit style; A vertex invokes `add_request()` with the destination vertex ID; all the requests are delivered after the request phase, and the vertices receiving any request will

be automatically involved, and a response value is produced by the user-provided function.

Propagation Channel

The last optimized channel is to speedup the convergence for a class of propagation-based algorithms. In these algorithms, typically, some vertices emit the initial labels, and in each of the following supersteps, vertices receiving the labels will perform some computation and may further propagate a new label to their outgoing neighbors. Since the propagation is between neighbors, such algorithms converge very slowly on graphs with large diameters.

The design of this channel is inspired by two existing techniques for improving the convergence speed. First, the GAS model [11] with an asynchronous execution mode can perform the crucial updates as early as possible without waiting for the global synchronization. Although this implementation is not feasible in our synchronous system, the high-level abstraction is suitable for describing such kind of computation. Second, the block-centric computation model [24, 26, 25] is an extension of Pregel which opens the partition to users, so that users can choose a suitable partition method and implement a block-level computation to perform the label propagation within a connected subgraph.

Our propagation channel combines the advantages of these two techniques: it provides a simplified GAS model which naturally describes such propagation-based computation, and its implementation works in a similar way as a block-level program to accelerate the label propagation. Therefore, users allocate a channel to obtain a performance gain without additional efforts on writing the block-level program.

Figure 3.7 describes the high-level model for the propagation channel as well as the execution logic in our implementation. Initially, each vertex is associated with a value and is set to active. Whenever having an active vertex u in the graph, it reads each incoming neighbors and the corresponding edges (if exists), and calculate a value a_i by a user-provided function f. Then, a combiner h updates the original vertex value u by each neighbor's a_i and returns a new vertex value u'. If the new value u' is different from the original value u, we activate all outgoing neighbors of u to propagate the update, and finally u is deactivated after being processed. The computation stops when

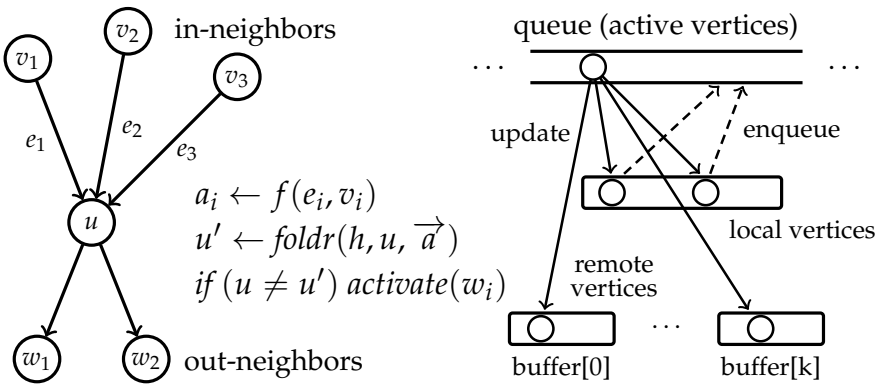


Figure 3.7: The propagation channel's high-level model and computation logic.

all the vertices are inactive. Note that we require h to be commutative, so that the order of combining a_i does not affect the result. Moreover, when any of the incoming neighbors of u is modified, u needs to read the modified vertex to update its own value, instead of recomputing the *foldr* by reading all its incoming neighbors' values.

This computation model is implemented by each worker performing a BFS-like traversal on the subgraph it holds. Starting from the initial setting, each worker propagates the values along the edges as far as possible. It updates the local vertices directly, but records the changes on remote vertices as messages. The buffer exchange is performed after no update is viable on any worker. After the remote updates triggered by messages, a new round of local propagation is performed. It terminates when all vertices have converged.

The last column of Table 3.2 shows the APIs of a simplified propagation channel without considering the edge weights (for saving space), so users provide a combiner to calculate the new vertex value. Each vertex adds its adjacent list to the channel via `add_edge()` and sets the initial value by `set_value()`, and in the next superstep, a vertex invokes `get_value()` to get the final value after the propagation converges. To make the best use of the propagation channel, users should properly partition the graph and attach the partition IDs to the vertex IDs.

Table 3.3: Datasets used in our evaluation

Dataset	$\|V\|$	$\|E\|$	avg. Deg
SubDomain	99.41M	1.94B	19.52
Twitter	41.65M	1.47B	70.51
Tree*	1.00B	1.00B	1.00
Chain*	1.00B	1.00B	1.00
RMAT*	400M	2.00B	5.00
Wikipedia	18.27M	172.31M	9.43

datasets marked with $*$ are synthetic.

3.3 Evaluation

The experiments are conducted on an Amazon EC2 cluster of 16 nodes (with instance type m5.2xlarge), each having 8 vCPUs and 32G memory. The connectivity between any pair of nodes in the cluster is 10Gb. The datasets are listed in Table 5.1 including both real-world graphs (Wikipedia[3], Twitter[4] and SubDomain[5]) and synthesized graphs (Chain, Tree and RMAT [37]). Graphs are converted to the required type (directed, undirected or weighted) for each algorithm in the experiments.

We select six representative algorithms in our evaluation, including PageRank (PR), Pointer-Jumping (PJ), Weakly Connected Component (WCC), SV algorithm (SV), Strongly Connected Component (SCC) and Minimum Spanning Forest (MSF). For comparison, we also present the results of our best-effort implementations in Pregel+ [21] and Blogel [26]. Both of them are typical Pregel implementations, where Pregel+ supports the request-respond paradigm and mirroring technique in two special modes (*reqresp* mode and *ghost* mode respectively) and Blogel supports the block-centric computation. All of these systems mentioned above as well as our channel-based system are implemented in C++ on top of the Hadoop Distributed File System (HDFS). The source code of our system can be accessed at `https://bitbucket.org/zyz915/pregel-channel`.

[3] http://konect.uni-koblenz.de/networks/dbpedia-link
[4] http://konect.uni-koblenz.de/networks/twitter_mpi
[5] http://webdatacommons.org/hyperlinkgraph/2012-08/download.html

3.3.1 Overhead of the channel mechanism

First, we evaluate the standard channels (the message passing channels and aggregator) in our system. Basically, rewriting a Pregel program into a channel-based version is just about replacing the matched send-receive pairs into the same channel's send/receive function. The message is chosen as small as possible, and we always use a combiner if applicable. We compare both implementations to see whether there is any overhead or benefits introduced by our channel mechanism.

The experiment results are presented in Table 3.4, where a straightforward rewriting achieves a speedup ranging from 1.16x to 4.16x among all the five algorithms on those datasets. For SCC and SV, we also observe a significant reduction on message size ranging from 23% to 62%.

Analysis. The channel mechanism itself can improve the performance, due to the following two reasons. First, our system allows users to specify a combiner to a channel whenever applicable, while in Pregel, we can specify a global combiner only when all the messages in the algorithm can use that combiner. This difference makes our SV and SCC more message-efficient, where the inapplicability of combiner in Pregel+ causes a 4.16x and 2.10x message size for SV and SCC respectively on Twitter.

Second, our channel-based system allows users to choose different message types for different channels, while in Pregel+, a global message type is chosen to serve all communication in the program. Then, the MSF (we refer to a particular version here [18]) is a typical example that uses heterogeneous messages in different phases of the algorithm. The largest message type is a 4-tuple of integer values for storing an edge, but the smallest one is just an `int`.

For the rest algorithms, there is no significant difference when implemented in two systems. Still, our system reduces the runtime of PR and WCC by up to 26% and 35% (using the `CombinedMessage` class), and for PJ (using the `DirectMessage` class) the number is 52%. We believe that the improvement is due to the choice of message interface (in particular the message iterator in `DirectMessage` instead of nested C++ vectors in Pregel+). Nevertheless, we show that our system implementation is reasonably efficient.

Table 3.4: Comparison of the basic implementation of graph algorithms in Pregel+ and channel-based system.

PJ	Chain		Tree		
	pregel	channel	pregel	channel	
runtime (s)	596.99	327.86	221.76	105.70	
msg (GB)	463.86	463.86	89.17	89.17	

SV	RMAT		SubDomain		Twitter	
	pregel	channel	pregel	channel	pregel	channel
runtime (s)	465.98	232.91	411.67	155.36	143.03	59.71
msg (GB)	212.64	112.49	298.62	71.82	115.96	28.53

PR	RMAT		SubDomain		Twitter	
	pregel	channel	pregel	channel	pregel	channel
runtime (s)	509.93	417.70	308.83	236.36	235.37	194.00
msg (GB)	413.03	413.03	160.99	160.99	128.81	128.81

WCC	RMAT		SubDomain		Twitter	
	pregel	channel	pregel	channel	pregel	channel
runtime (s)	65.36	42.31	54.61	41.02	22.58	17.47
msg (GB)	37.83	37.83	19.65	19.65	9.26	9.26

SCC	RMAT		SubDomain		Twitter	
	pregel	channel	pregel	channel	pregel	channel
runtime (s)	266.52	116.24	345.38	382.68	99.99	56.64
msg (GB)	118.56	91.09	132.07	62.99	77.53	45.79

MSF	RMAT		SubDomain		Twitter	
	pregel	channel	pregel	channel	pregel	channel
runtime (s)	547.98	319.86	200.07	138.71	138.92	119.27
msg (GB)	438.41	400.11	173.18	161.53	123.36	117.50

3.3.2 Effectiveness of optimized channels

Here, we evaluate the efficiency of our optimized channels against the message passing channels using the applications that each kind of channel is applicable. In this part, we choose rather simple algorithms, so that we can clearly see how optimized channels can improve the performance in different scenarios.

Scatter-combine channel

PageRank is a typical graph algorithm that can be optimized by the scatter-combine channel. We test Pregel+'s basic implementation, Pregel+'s *ghost* mode (a.k.a. the mirroring technique [21]), the standard channel version (Figure 3.1) and the scatter-combine channel version. For Pregel+'s mirroring technique, we set the threshold to 16 in all cases.

The experiment results are presented in the upper part of Table 3.5. The basic mode of Pregel+ and our standard version are close in both execution time and message size, while the scatter-combine channel achieves a speedup ranging from 3.39x to 3.50x and reduces roughly one third of the message size. Pregel+'s ghost mode use less messages, but the execution time (including the preprocessing time) is not reduced significantly.

Analysis. The improvement on execution time clearly shows the effectiveness of the scatter-combine channel. As explained in Section 3.2.4, it can generate the combined messages by a linear scan of the edges, while Pregel+'s basic mode and the `CombinedMessages` have to use hash table or sorting in every superstep. The reduction on total message size is explained by the removal of redundant transmission of vertices' identifiers.

All these three programs use the *receiver-centric* message combining (for high-degree receiver), while Pregel+'s mirroring technique has the *sender-centric* message combining to further reduce the messages. However, such method is computational intensive and the overall computational cost is higher. We show that the computational cost in message processing is a major problem in some algorithms, and our scatter-combine achieves better performance than existing approaches.

Request-respond channel

We consider the pointer-jumping algorithm (which is also part of the SV algorithm) as a minimum example that uses the request-respond paradigm. Given a (forest of) rooted tree, each vertex initially knows its parent and tries to find the root of the tree it belongs to. We test Pregel+'s basic implementation, Pregel+'s *reqresp* mode (which is the original implementation of the request-respond paradigm [21]), the standard channel version and the scatter-combine channel version. We use two types of graphs, a randomly generated tree and a chain. Vertices are randomly assigned to workers.

Table 3.5: Experiment results for each optimized channel.

Scatter-Combine channel using PR				
Program	SubDomain		Twitter	
	runtime	message	runtime	message
pregel+ (basic)	308.83	160.99	235.37	128.81
pregel+ (ghost)	353.77	152.50	256.95	111.28
channel (basic)	236.36	160.99	194.00	128.81
channel (scatter)	88.18	109.12	69.46	87.31
Request-Respond channel using PJ				
Program	Tree		Chain	
	runtime	message	runtime	message
pregel+ (basic)	221.76	89.17	596.99	463.86
pregel+ (reqresp)	342.44	29.88	4279.06	336.27
channel (basic)	105.70	89.17	327.86	463.86
channel (reqresp)	50.29	19.92	328.87	224.18
Propagation channel using WCC				
Program	Wikipedia		Wikipedia (P)	
	runtime	message	runtime	message
pregel+ (basic)	16.96	2.85	15.31	0.49
blogel	20.39	1.11	5.10	0.11
channel (basic)	15.67	2.85	15.85	0.49
channel (prop.)	8.64	1.66	3.05	0.17

The middle part of Table 3.5 summarizes the results on the two graphs. Without the request-respond optimization, the standard implementations in the two systems use exactly the same number of messages, but ours runs 2.10x faster on a chain 1.82x faster on a randomly generated tree. Contrary to our expectation, Pregel+'s *reqresp* mode has a negative effect on the execution time, although the message size indeed decreases. Our implementation of the request-respond paradigm shows reasonable results, which runs faster on a randomly generated tree, and is as good as an ordinary implementation when tree degrades to a chain. Compared to Pregel+'s *reqresp* mode, our implementation constantly reduces the message size by 33%, and achieves a significant performance gain (up to 13.01x) on the Chain.

Analysis. Although sharing the same idea, the implementations of the request-respond paradigm in our system and Pregel+ are different, which we believe is the main reason that makes our implementation better in both runtime and message size. The

request-respond channel works better on Tree, since it easily generates high-degree vertices during the computation. For Chain, there is actually no high-degree vertex until the final stage of the algorithm, but it does not compensate the computational overhead in the channel implementation.

We also observe that, in real algorithms like SV (Section 3.1.3), we are actually dealing with a dynamic forest, where the finding of the root vertex root is fused with the tree merging. In this special case, Pregel+'s *reqresp* mode can still make an improvement (see Table 3.6). Nevertheless, we verify that our implementation of the request-respond technique is reasonably effective, and is faster than the one in Pregel+.

Propagation channel

We consider the HCC algorithm [38] as a suitable example for using this optimization, which finds the weakly connected component (WCC) of a directed graph. In this experiment, we present both the results on the original Wikipedia graph and the partitioned graph by METIS [39]. We also add the Blogel version here since the block-centric model is applicable [26]. We choose METIS since it requires no additional knowledge of the graph.

The experiment results are presented in the bottom part of Table 3.5. First, the Pregel+ program and a standard channel version in our system are very close in both execution time and message size. The block-centric version in Blogel works slightly worse on the original graph, but achieves roughly 3x faster when the input graph is properly partitioned. Our propagation channel version works consistently better than all other implementations in terms of execution time on both graphs (1.67x faster than Blogel). The number of messages used in the propagation channel version is the same as the Blogel version, but the message size in Blogel is 33% less due to its special treatment of partition information. Nevertheless, running WCC on partitioned graph is not message intensive.

Analysis. A partitioner reduces the communication cost between the workers, but for the standard WCCs (program 1 and 3), it still takes a large number of supersteps to converge, so the execution time is not reduced. Both of Blogel and our propagation channel use a block-level program to speedup the convergence and our system outperforms Blogel slightly.

Table 3.6: Experiment results of the SV implementations using different combinations of channels.

Program	SubDomain		Twitter	
	runtime	message	runtime	message
1-pregel+ (basic)	411.67	298.62	143.03	115.96
2-pregel+ (reqresp)	174.67	66.17	74.20	29.12
3-channel (basic)	155.36	71.82	59.71	28.53
4-channel (reqresp)	128.96	59.74	53.16	24.86
5-channel (scatter)	75.45	44.69	31.86	18.15
6-channel (both)	51.59	32.60	24.94	14.49

It is also noteworthy that, WCC's standard implementation is simply an iterative neighborhood communication that needs around 10 lines of code for the `compute()` function. While the Blogel version requires users to additionally write a block-level computation of more than 100 lines of code[6], switching to the propagation channel in our system is much easier. It is clear that our system achieves both conciseness and efficiency compared to the block-centric model.

3.3.3 Combination of channels

In this part, we verify the multiple performance issues in the SV (see discussions in Section 3.1.3) by running the programs using different combination of channels in our system. We show that a combination of properly chosen channels can finally lead to much better performance. To cover all the special channels we have, we also present the experiment results of the Min-Label algorithm [21] for finding Strongly Connected Components (SCCs).

The SV Algorithm

According to the previous discussion, the request-respond channel and the scatter-combine channel are applicable in the algorithm implementation. We thus have four SV programs in our system covering all the combination of the two optimized channels. For comparison, we also give the result of our best-effort implementation in Pregel+'s *basic* and *reqresp* modes.

[6] http://www.cse.cuhk.edu.hk/blogel/code/apps/block/hashmin/block.zip

Table 3.7: Experiment results of the Min-Label algorithm

Program	Wikipedia		Wikipedia (P)	
	runtime	message	runtime	message
1-pregel+ (basic)	52.15	9.85	50.51	2.70
2-channel (basic)	61.89	4.98	67.84	1.29
3-channel (prop.)	31.37	4.42	13.96	1.12

The results are presented in Table 3.6. As expected, the basic version (program 3) without using any specialized channel is the slowest, and the fully optimized version (program 6) takes only one third of the execution time. Furthermore, using either of the request-respond channel (program 4) or the scatter-combine channel (program 5) can lead to a decent improvement on both graphs. Pregel+'s basic mode runs extremely slowly, which is mainly due to the inapplicability of the combiner optimization. Then, even with the request-respond paradigm (in which the combiner optimization is enabled), Pregel+ is still slower than our unoptimized version on both graphs.

Analysis. The experiment clearly verifies the multiple performance issues in the SV implementation. Even with the request-respond optimization, the SV algorithm still suffers the heavy communication cost, since the redundancies in the neighborhood communication become the major problem. Our system combines all the optimizations and makes the algorithm work consistently well.

Min-Label Algorithm

Strongly connected component (SCC) is a fundamental problem in graph theory and it is widely used in practice to reveal the properties of the graphs. A typical Min-Label algorithm [21] for finding SCCs in Pregel is already complex which is an iterative algorithm where the main iteration contains four subroutines, including the removal of trivial SCCs, forward/backward label propagation, SCC recognization and relabeling. The algorithm suffers the problem of low convergence speed.

Our system offers the `Propagation` channel for the forward/backward label propagation, which achieves a 2x speedup on Wikipedia, and a nearly 4x faster on partitioned Wikipedia (see Table 3.7). This optimization is not possible in any of the existing system.

3.4 Related work

Google's Pregel [1] is the first specific in-memory system for distributed graph processing. It adopts the Bulk-Synchronous Parallel (BSP) model [9] with explicit messages to let users implement graph algorithms in a vertex-centric way. The core design of Pregel has been widely adopted by many open-source frameworks [2, 3], and most of them inherit the monolithic message passing interface, meaning that the messages of different purposes are mixed and indistinguishable for the system. As an attempt for optimizing communication patterns, Pregel+ extends Pregel with additional interfaces (in particular, the *reqresp* and the *ghost* mode), but it is less flexible since the two modes cannot be composed and adding optimizations is inconvenient.

To support intuitive message slicing in Pregel-like systems, Telos [40] proposes a layered architecture where interleaving tasks are implemented as separate *Protocols*, each having a user-defined `compute()` function with a dedicated message buffer. However, it lacks an essential feature for optimization that users cannot modify the implementation of the message buffer. Husky [34] is a general-purpose distributed framework with the channel interface, and it supports primitives like *pull*, *push* and *migrate* and *asynchronous updates* to combine the strength of graph-parallel and machine learning systems. We extend this idea for composing optimizations in graph-parallel system and propose our optimized channels for three common performance issues.

There has been much research studying the optimizations on Pregel-like systems, and our optimized channels draw inspiration from this line of research, such as the sender-side message combining (a.k.a. vertex-replication, mirroring) [23, 22, 29, 21], the request-respond paradigm [21], the block-centric model [24, 26, 25] and so on. In particular, our scatter-combine channel recognizes the static messaging pattern and reduces the computational cost as well as message size by preprocessing, which is novel and turns out to be effective for communication-intensive algorithms like PageRank and SV. We also demonstrate how complex algorithms like SV and SCC can be optimized by such technique, while most existing systems only focus on rather simple algorithms.

Apart from Pregel, there are graph-parallel systems that use high-level models to organize the computation and communication, which brings more opportunities for op-

timization. For example, the Gather-Apply-Scatter (GAS) model (used by GraphLab [41], PowerGraph [11] and PowerLyra [12]) is a typical one that describes a vertex-program by three functions, and the scatter-combine model (used by Graphine [42]) fuses the scatter and gather operations, resulting a more compact two-phase model. Our channel mechanism shares the same spirit; through the channels, we can equip a system with even more abstractions, so that users can choose whatever suitable for their algorithms.

There are also graph systems using a functional interface with high-level primitives to manipulate the entire graph, such as GraphX [43] (a library on top of Apache Spark [44]) and its extension HelP [45]. However, their primitives are hard to compose. Furthermore, experiment results [34] show that they are less efficient than other systems even on simple algorithms like PageRank. Sparse-matrix based frameworks (e.g. the CombBLAS [4] and PEGASUS[38]) are also popular for handling graphs which provide linear algebra primitives, but the lack of graph semantics makes it hard for deep optimizations.

4

Domain-Specific Languages

In this chapter, we propose two domain-specific language for simplifying the development of large-scale graph applications. We first present the Palgol language for high-level vertex-centric programming. Compared to Pregel's original model, we introduce *remote access* – reading or writing the other vertices' states – to hide the error-prone message passing, resulting in a more concise and flexible language that can express many graph algorithms. Case studies using popular graph applications are made to show the convenience of Palgol. Second, we present the SQL-core language which uses a relational model to present graph computations. Compared to the vertex-centric programming model, this language remains a global view of the graph computation, making a class of optimizations feasible to detect and apply by the compiler. Finally, we give a picture on how these two languages can be further combined, resulting a DSL with both user-friendly programming interface and a compiler to enable powerful optimizations to ensure high performance.

4.1 Palgol: the Pregel algorithmic language

We first introduce its programming model in which algorithm is decomposed into atomic vertex-centric computations and high-level combinators, and a vertex can access the entire graph through the references it stores locally. Next we present Palgol's syntax and semantics (Section 4.1.2). Finally we use two representative examples — the Shiloach-Vishkin connected component algorithm (Section 4.1.3) and the list ranking algorithm (Section 4.1.3) — to demonstrate how Palgol can concisely describe vertex-centric algorithms with dynamic internal structures using remote access.

4.1.1 The high-level model

The high-level model we propose uses remote reads and writes instead of message passing to allow programmers to describe vertex-centric computation more intuitively. Moreover, the model remains close to the Pregel computation model, in particular keeping the vertex-centric paradigm and barrier synchronization, making it possible to automatically derive a valid and efficient Pregel implementation from an algorithm description in this model, and in particular arrange remote reads and writes without data conflicts.

In our high-level model, the computation is constructed from some basic components which we call *algorithmic supersteps*. An algorithmic superstep is a piece of vertex-centric computation which takes a graph containing a set of vertices with local states as input, and outputs the same set of vertices with new states. Using algorithmic supersteps as basic building blocks, two high-level operations *sequence* and *iteration* can be used to glue them together to describe more complex vertex-centric algorithms that are iterative and/or consist of multiple computation stages: the *sequence* operation concatenates two algorithmic supersteps by taking the result of the first step as the input of the second one, and the *iteration* operation repeats a piece of vertex-centric computation until some termination condition is satisfied.

The distinguishing feature of algorithmic supersteps is remote access. Within each algorithmic superstep (illustrated in Figure 4.1), all vertices compute in parallel, performing the same computation specified by programmers. A vertex can read the fields of any vertex in the input graph; it can also write to arbitrary vertices to modify

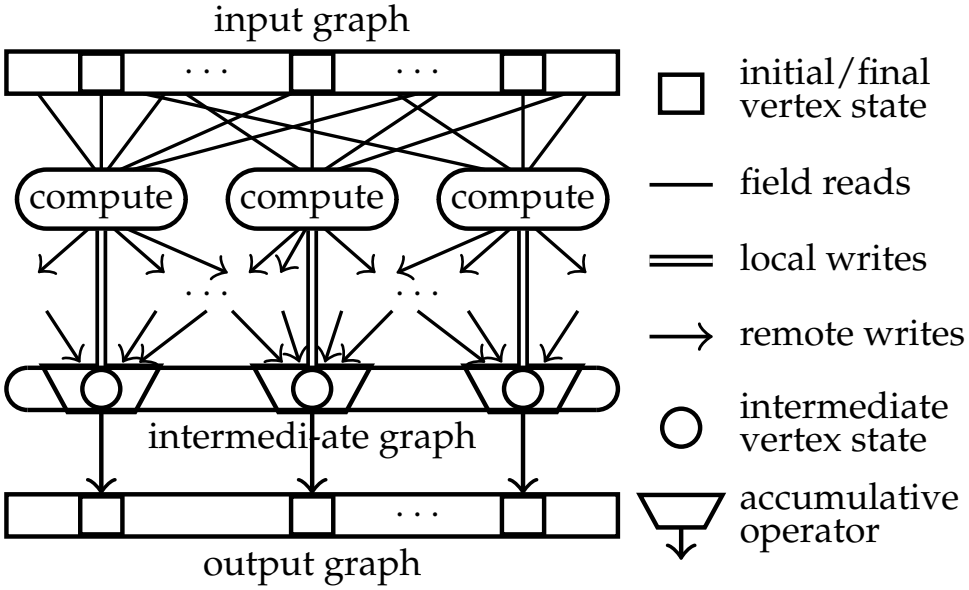


Figure 4.1: In an algorithmic superstep, every vertex performs local computation (including field reads and local writes) and remote updating in order.

their fields, but the writes are performed on a separate graph rather than the input graph (so there are no read-write conflicts). We further distinguish *local writes* and *remote writes* in our model: local writes can only modify the current vertex's state, and are first performed on an intermediate graph (which is initially a copy of the input graph); next, remote writes are propagated to the destination vertices to further modify their intermediate states. Here, a remote write consists of a remote field, a value and an "accumulative" assignment (like += and |=), and that field of the destination vertex is modified by executing the assignment with the value on its right-hand side. We choose to support only accumulative assignments so that the order of performing remote writes does not matter.

More precisely, an algorithmic superstep is divided into two phases:

- a *local computation* (LC) phase, in which a copy of the input graph is created as the intermediate graph, and then each vertex can read the state of any vertex in the input graph, perform local computation, and modify its own state in the intermediate graph, and
- a *remote updating* (RU) phase, in which each vertex can modify the states of any vertices in the intermediate graph by sending remote writes. After all remote

writes are processed, the intermediate graph is returned as the output graph.

Among these two phases, the RU phase is optional, in which case the intermediate graph produced by the LC phase is used directly as the final result.

4.1.2 An overview of Palgol

Next we present Palgol, whose design follows the high-level model we introduced above. Figure 4.2 shows the essential part of Palgol's syntax. As described by the syntactic category *step*, an algorithmic superstep in Palgol is a code block enclosed by "**for** *var* **in V**" and "**end**", where *var* is a variable name that can be used in the code block for referring to the current vertex (and **V** stands for the set of vertices of the input graph). Such steps can then be composed (by sequencing) or iterated until a termination condition is met (by enclosing them in "**do**" and "**until** ..."). Palgol supports several kinds of termination condition, but in this thesis we focus on only one kind of termination condition called *fixed point*, since it is extensively used in many algorithms. The semantics of fixed-point iteration is iteratively running the program enclosed by **do** and **until**, until the specified fields stabilize.

Corresponding to an algorithmic superstep's remote access capabilities, in Palgol we can read a field of an arbitrary vertex using a global field access expression of the form *field* [*exp*], where *field* is a user-specified field name and *exp* should evaluate to a vertex id. Such expression can be updated by local or remote assignments, where an assignment to a remote vertex should always be accumulative and prefixed with the keyword **remote**. One more thing about remote assignments is that they take effect only in the RU phase (after the LC phase), regardless of where they occur in the program.

There are some predefined fields that have special meaning in our language. **Nbr** is the edge list in undirected graphs, and **In** and **Out** respectively store incoming and outgoing edges for directed graphs. Essentially, these are normal fields of a predefined type for representing edges, and most importantly, the compiler assumes a form of symmetry on these fields (namely that every edge is stored consistently on both of its end vertices), and uses the symmetry to produce more efficient code.

The rest of the syntax for Palgol steps is similar to an ordinary programming language. Particularly, we introduce a specialized pair type (expressions in the

$$
\begin{array}{lcll}
\mathit{prog} & ::= & \mathit{step} \mid \mathit{prog}_1 \ldots \mathit{prog}_n \mid \mathit{iter} & \\
\mathit{iter} & ::= & \textbf{do} \langle\ \mathit{prog}\ \rangle\ \textbf{until fix}\ [\ \mathit{field}_1, \ldots, \mathit{field}_n\ ] & \\
\mathit{step} & ::= & \textbf{for}\ \mathit{var}\ \textbf{in V} \langle\ \mathit{block}\ \rangle\ \textbf{end} & \\
\mathit{block} & ::= & \mathit{stmt}_1 \ldots \mathit{stmt}_n & \\
\mathit{stmt} & ::= & \textbf{if}\ \mathit{exp} \langle\ \mathit{block}\ \rangle \mid \textbf{if}\ \mathit{exp} \langle\ \mathit{block}\ \rangle\ \textbf{else} \langle\ \mathit{block}\ \rangle & \\
 & \mid & \textbf{for}\ (\mathit{var} \leftarrow \mathit{exp}) \langle\ \mathit{block}\ \rangle & \\
 & \mid & \textbf{let}\ \mathit{var} = \mathit{exp} & \\
 & \mid & \textbf{local}_{\mathit{opt}}\ \mathit{field}\ [\ \mathit{var}\ ]\ \mathit{op}_{\mathit{local}}\ \mathit{exp} & \text{– local write} \\
 & \mid & \textbf{remote}\ \mathit{field}\ [\ \mathit{exp}\ ]\ \mathit{op}_{\mathit{remote}}\ \mathit{exp} & \text{– remote write} \\
\mathit{exp} & ::= & \mathit{int} \mid \mathit{float} \mid \mathit{var} \mid \textbf{true} \mid \textbf{false} \mid \textbf{inf} & \\
 & \mid & \textbf{fst}\ \mathit{exp} \mid \textbf{snd}\ \mathit{exp} \mid (\mathit{exp}, \mathit{exp}) & \\
 & \mid & \mathit{exp}.\textbf{ref} \mid \mathit{exp}.\textbf{val} \mid \{\mathit{exp}, \mathit{exp}\} \mid \{\mathit{exp}\} & \text{– specialized pair} \\
 & \mid & \mathit{exp}\ ?\ \mathit{exp}\ :\ \mathit{exp} \mid (\ \mathit{exp}\ ) \mid \mathit{exp}\ \mathit{op}_b\ \mathit{exp} \mid \mathit{op}_u\ \mathit{exp} & \\
 & \mid & \mathit{field}\ [\ \mathit{exp}\ ] & \text{– global field access} \\
 & \mid & \mathit{func}_{\mathit{opt}}\ [\ \mathit{exp} \mid \mathit{var} \leftarrow \mathit{exp}, \mathit{exp}_1, \ldots, \mathit{exp}_n\ ] & \\
\mathit{func} & ::= & \textbf{maximum} \mid \textbf{minimum} \mid \textbf{sum} \mid \ldots &
\end{array}
$$

Figure 4.2: Essential part of Palgol's syntax. Palgol is indentation-based, and two special tokens '⟨' and '⟩' are introduced to delimit indented blocks.

form of {*exp*, *exp*}) for representing a reference with its corresponding value (e.g., an edge in a graph), and use **.ref** and **.val** respectively to access the reference and the value respectively, to make the code easy to read. Some functional programming constructs are also used here, like let-binding and list comprehension. There is also a foreign function interface that allows programmers to invoke functions written in a general-purpose language, but we omit the detail from the paper.

4.1.3 Case studies

Next, we use two examples to show how to use Palgol to implement graph algorithms.

The Shiloach-Vishkin Connected Component Algorithm

Here is our first representative Palgol example: the *Shiloach-Vishkin (SV) connected component algorithm* [21], which can be expressed as the Palgol program in Figure 4.3. A traditional HashMin connected component algorithm [21] based on neighborhood communication takes time proportional to the input graph's diameter, which can be large in real-world graphs. In contrast, the SV algorithm can calculate the connected

components of an undirected graph in a logarithmic number of supersteps; to achieve this fast convergence, the capability of accessing data on non-neighboring vertices is essential.

In the SV algorithm, the connectivity information is maintained using the classic disjoint set data structure [35]. Specifically, the data structure is a forest, and vertices in the same tree are regarded as belonging to the same connected component. Each vertex maintains a parent pointer that either points to some other vertex in the same connected component, or points to itself, in which case the vertex is the root of a tree. We henceforth use $D[u]$ to represent this pointer for each vertex u. The SV algorithm is an iterative algorithm that begins with a forest of n root nodes, and in each step it tries to discover edges connecting different trees and merge the trees together. In a vertex-centric way, every vertex u performs one of the following operations depending on whether its parent $D[u]$ is a root vertex:

- **tree merging:** if $D[u]$ is a root vertex, then u chooses one of its neighbors' current parent (to which we give a name t), and makes $D[u]$ point to t if $t < D[u]$ (to guarantee the correctness of the algorithm). When having multiple choices in choosing the neighbors' parent p, or when different vertices try to modify the same parent vertex's pointer, the algorithm always uses the "minimum" as the tiebreaker for fast convergence.

- **pointer jumping:** if $D[u]$ is not a root vertex, then u modifies its own pointer to its current "grandfather" ($D[u]$'s current pointer). This operation reduces u's distance to the root vertex, and will eventually make u a direct child of the root vertex so that it can perform the above tree merging operation.

The algorithm terminates when all vertices' pointers do not change after an iteration, in which case all vertices point to some root vertex and no more tree merging can be performed. Readers interested in the correctness of this algorithm are referred to the original paper [21] for more details.

The implementation of this algorithm is complicated, which contains roughly 120 lines of code[1] for the *compute()* function alone. Even for detecting whether the parent vertex $D[u]$ is a root vertex for each vertex u, it has to be translated into three

[1] http://www.cse.cuhk.edu.hk/pregelplus/code/apps/basic/svplus.zip

```
1  for u in V
2    D[u] := u
3  end
4  do
5    for u in V
6      if (D[D[u]] == D[u])
7        let t = minimum [ D[e.ref] | e <- Nbr[u] ]
8        if (t < D[u])
9          remote D[D[u]] <?= t
10     else
11       D[u] := D[D[u]]
12   end
13 until fix[D]
```

Figure 4.3: The SV algorithm in Palgol

supersteps containing a query-reply conversation between each vertex and its parent. In contrast, the Palgol program in Figure 4.3 can describe this algorithm concisely in 13 lines, due to the declarative remote access syntax. This piece of code contains two steps, where the first one (lines 1–3) performs simple initialization, and the other (lines 5–12) is inside an iteration as the main computation. We also use the field D to store the pointer to the parent vertex. Let us focus on line 6, which checks whether u's parent is a root. Here we simply check $D[D[u]] == D[u]$, i.e., whether the pointer of the parent vertex $D[D[u]]$ is equal to the parent's id $D[u]$. This expression is completely declarative, in the sense that we only specify what data is needed and what computation we want to perform, instead of explicitly implementing the message passing scheme.

The rest of the algorithm can be straightforwardly associated with the Palgol program. If u's parent is a root, we generate a list containing all neighboring vertices' parent id ($D[e.\mathbf{ref}]$), and then bind the minimum one to the variable t (line 7). Now t is either **inf** if the neighbor list is empty or a vertex id; in both cases we can use it to update the parent's pointer (lines 8–9) via a remote assignment. One important thing is that the parent vertex ($D[u]$) may receive many remote writes from its children, where only one of the children providing the minimum t can successfully perform the updating. Here, the statement `a <?= b` is an accumulative assignment, whose meaning is the same as `a := min(a, b)`. Finally, for the **else** branch, we (locally)

```
1  for u in V
2    Sum[u] := Val[u]
3  end
4  do
5    for u in V
6      if (Pred[Pred[u]] != Pred[u])
7        Sum[u] += Sum[Pred[u]]
8        Pred[u] := Pred[Pred[u]]
9    end
10 until fix[Pred]
```

Figure 4.4: The list ranking program

assign u's grandparent's id to u's D field.

The List Ranking Algorithm

Another example is the *list ranking* algorithm, which also needs communication over a dynamic structure during computation. Consider a linked list L with n elements, where each element u stores a value $val(u)$ and a link to its predecessor $pred(u)$. At the head of L is a virtual element v such that $pred(v) = v$ and $val(v) = 0$. For each element u in L, define $sum(u)$ to be the sum of the values of all the elements from u to the head (following the predecessor links). The list ranking problem is to compute $sum(u)$ for each element u. If $val(u) = 1$ for every vertex u in L, then $sum(u)$ is simply the rank of u in the list. List ranking can be solved using a typical pointer-jumping algorithm in parallel computing with a strong performance guarantee. Yan et al. [21] demonstrated how to compute the pre-ordering numbers for all vertices in a tree in $O(\log n)$ supersteps using this algorithm, as an internal step to compute bi-connected components (BCC).[2]

We give the Palgol implementation of list ranking in Figure 4.4 (which is a 10-line program, whereas the Pregel implementation[3] contains around 60 lines of code). $Sum[u]$ is initially set to $Val[u]$ for every u at line 2; inside the fixed-point iteration (lines 5–9), every u moves $Pred[u]$ toward the head of the list and updates $Sum[u]$ to maintain

[2]BCC is a complicated algorithm, whose efficient implementation requires constructing an intermediate graph, which is currently beyond Palgol's capabilities. Palgol is powerful enough to express the rest of the algorithm, however.

[3]http://www.cse.cuhk.edu.hk/pregelplus/code/apps/basic/bcc.zip

the invariant that $Sum[u]$ stores the sum of a sublist from itself to the successor of $Pred[u]$. Line 6 checks whether u points to the virtual head of the list, which is achieved by checking $Pred[Pred[u]] == Pred[u]$, i.e., whether the current predecessor $Pred[u]$ points to itself. If the current predecessor is not the head, we add the sum of the sublist maintained in $Pred[u]$ to the current vertex u, by reading $Pred[u]$'s *Sum* and *Pred* fields and modifying u's own fields accordingly. Note that since all the reads are performed on a snapshot of the input graph and the assignments are performed on an intermediate graph, there is no need to worry about data dependencies.

4.2 Compiling Palgol to Pregel

In this section, we present the compiling algorithm to transform Palgol to Pregel. The task overall is complicated and highly technical, but the most challenging problem is how to translate chain access (like $D[D[u]]$) into Pregel's message passing model. We describe the compilation of chain access in Section 4.2.1, and then the compilation of a Palgol step in Section 4.2.2, and finally how to combine Palgol steps using sequence and iteration in Section 4.2.3.

4.2.1 Compiling remote reads

Our compiler currently recognizes two forms of remote reads. The first form is *chain access* expressions like $D[D[u]]$. The second form is *neighborhood access* where a vertex may use chain access to acquire data from *all* its neighbors, and this can be described using the list comprehension (e.g., line 7 in Figure 4.3) or for-loop syntax in Palgol. The combination of these two remote read patterns is already sufficient to express quite a wide range of practical Pregel algorithms. Here we only present the compilation of chain access, which is novel, while the compilation of neighborhood access is similar to what has been done in Fregel.

Definition and challenge of compiling:

A chain access is a consecutive field access expression starting from the current vertex. As an example, supposing that the current vertex is u, and D is a field for storing a vertex id, then $D[D[u]]$ is a chain access expression, and so is $D[D[D[D[u]]]]$ (which

we abbreviate to $D^4[u]$ in the rest of this section). Generally speaking, there is no limitation on the depth of a chain access or the number of fields involved in the chain access.

As a simple example of the compilation, to evaluate $D[D[u]]$ on every vertex u, a straightforward scheme is a request-reply conversation which takes two rounds of communication: in the first superstep, every vertex u sends a request to (the vertex whose id is) $D[u]$ and the request message should contain u's own id; then in the second superstep, those vertices receiving the requests should extract the sender's ids from the messages, and reply its D field to them.

When the depth of such chain access increases, it is no longer trivial to find an efficient scheme, where efficiency is measured in terms of the number of supersteps taken. For example, to evaluate $D^4[u]$ on every vertex u, a simple query-reply method takes six rounds of communication by evaluating $D^2[u]$, $D^3[u]$ and $D^4[u]$ in turn, each taking two rounds, but the evaluation can actually be done in only three rounds with our compilation algorithm, which is not based on request-reply conversations.

Logic system for compiling chain access:

The key insight leading to our compilation algorithm is that we should consider not only the expression to evaluate but also the vertex on which the expression is evaluated. To use a slightly more formal notation (inspired by Halpern and Moses [46]), we write $\forall u.\, \mathrm{K}_{v(u)}\, e(u)$, where $v(u)$ and $e(u)$ are chain access expressions starting from u, to describe the state where every vertex $v(u)$ "knows" the value of the expression $e(u)$; then the goal of the evaluation of $D^4[u]$ can be described as $\forall u.\, \mathrm{K}_u\, D^4[u]$. Having introduced the notation, the problem can now be treated from a logical perspective, where we aim to search for a derivation of a target proposition from a few axioms.

There are three axioms in our logic system:

1. $\forall u.\, \mathrm{K}_u\, u$
2. $\forall u.\, \mathrm{K}_u\, D[u]$
3. $(\forall u.\, \mathrm{K}_{w(u)}\, e(u)) \wedge (\forall u.\, \mathrm{K}_{w(u)}\, v(u)) \implies \forall u.\, \mathrm{K}_{v(u)}\, e(u)$

The first axiom says that every vertex knows its own id, and the second axiom says every vertex can directly access its local field D. The third axiom encodes message

$$
\begin{array}{llll}
(\forall u.\, \mathrm{K}_u \quad u \quad) \wedge (\forall u.\, \mathrm{K}_u \quad D[u]\,) & \Longrightarrow & \forall u.\, \mathrm{K}_{D[u]} \quad u \\
(\forall u.\, \mathrm{K}_{D[u]} \quad u \quad) \wedge (\forall u.\, \mathrm{K}_{D[u]} \quad D^2[u]) & \Longrightarrow & \forall u.\, \mathrm{K}_{D^2[u]} \quad u \\
(\forall u.\, \mathrm{K}_{D[u]} \quad D^2[u]) \wedge (\forall u.\, \mathrm{K}_{D[u]} \quad u \quad) & \Longrightarrow & \forall u.\, \mathrm{K}_u \quad D^2[u] \\
(\forall u.\, \mathrm{K}_{D^2[u]} \quad D^4[u]) \wedge (\forall u.\, \mathrm{K}_{D^2[u]} \quad u \quad) & \Longrightarrow & \forall u.\, \mathrm{K}_u \quad D^4[u]
\end{array}
$$

Figure 4.5: A derivation of $\forall u.\, \mathrm{K}_u\, D^4[u]$

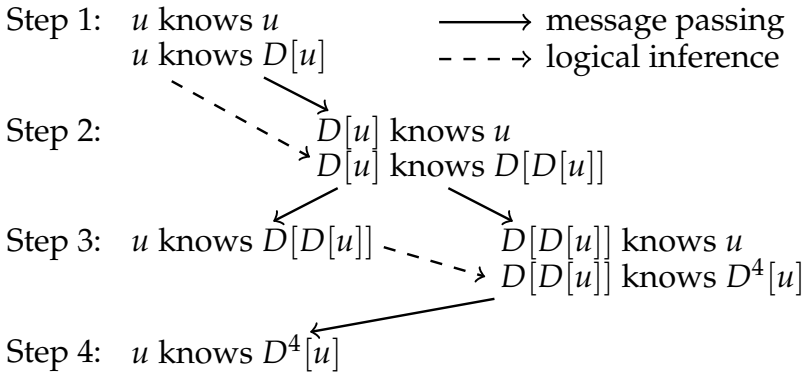


Figure 4.6: Interpretation of the derivation of $\forall u.\, \mathrm{K}_u\, D^4[u]$

passing: if we want every vertex $v(u)$ to know the value of the expression $e(u)$, then it suffices to find an intermediate vertex $w(u)$ which knows both the value of $e(u)$ and the id of $v(u)$, and thus can send the value to $v(u)$. As an example, Figure 4.5 shows the solution generated by our algorithm to solve $\forall u.\, \mathrm{K}_u\, D^4[u]$, where each line is an instance of the message passing axiom.

Figure 4.6 is a direct interpretation of the implications in Figure 4.5. To reach $\forall u.\, \mathrm{K}_u\, D^4[u]$, only three rounds of communication are needed. Each solid arrow represents an invocation of the message passing axiom in Figure 4.5, and the dashed arrows represent two logical inferences, one from $\forall u.\, \mathrm{K}_u\, D[u]$ to $\forall u.\, \mathrm{K}_{D[u]}\, D^2[u]$ and the other from $\forall u.\, \mathrm{K}_u\, D^2[u]$ to $\forall u.\, \mathrm{K}_{D^2[u]}\, D^4[u]$.

The derivation of $\forall u.\, \mathrm{K}_u\, D^4[u]$ is not unique, and there are derivations that correspond to inefficient solutions — for example, there is also a derivation for the six-round solution based on request-reply conversations. However, when searching for derivations, our algorithm will minimize the number of rounds of communication, as explained below.

The compiling algorithm:

Initially, the algorithm sets as its target a proposition $\forall u.\, \mathrm{K}_{v(u)}\, e(u)$, for which a derivation is to be found. The key problem here is to choose a proper $w(u)$ so that, by applying the message passing axiom backwards, we can get two potentially simpler new target propositions $\forall u.\, \mathrm{K}_{w(u)}\, e(u)$ and $\forall u.\, \mathrm{K}_{w(u)}\, v(u)$ and solve them respectively. The range of such choices is in general unbounded, but our algorithm considers only those simpler than $v(u)$ or $e(u)$. More formally, we say that a is a *subpattern* of b, written $a \preceq b$, exactly when b is a chain access starting from a. For example, u and $D[u]$ are subpatterns of $D[D[u]]$, while they are all subpatterns of $D^3[u]$. The range of intermediate vertices we consider is then $\mathrm{Sub}(e(u), v(u))$, where Sub is defined by

$$\mathrm{Sub}(a, b) = \{\, c \mid c \preceq a \text{ or } c \prec b \,\}$$

We can further simplify the new target propositions with the following function before solving them:

$$\mathit{generalize}(\forall u.\, \mathrm{K}_{a(u)}\, b(u)) = \begin{cases} \forall u.\, \mathrm{K}_u\, (b(u)/a(u)) & \text{if } a(u) \preceq b(u) \\ \forall u.\, \mathrm{K}_{a(u)}\, b(u) & \text{otherwise} \end{cases}$$

where $b(u)/a(u)$ denotes the result of replacing the innermost $a(u)$ in $b(u)$ with u. (For example, $A[B[C[u]]]/C[u] = A[B[u]]$.) This is justified because the original proposition can be instantiated from the new proposition. (For example, $\forall u.\, \mathrm{K}_{C[u]}\, A[B[C[u]]]$ can be instantiated from $\forall u.\, \mathrm{K}_u\, A[B[u]]$.)

It is now possible to find an optimal solution with respect to the following inductively defined function *step*, which calculates the number of rounds of communication for a proposition:

$$\begin{aligned} \mathit{step}(\forall u.\, \mathrm{K}_u\, u) &= 0 \\ \mathit{step}(\forall u.\, \mathrm{K}_u\, D[u]) &= 0 \\ \mathit{step}(\forall u.\, \mathrm{K}_{v(u)}\, e(u)) &= 1 + \min_{w(u) \in \mathrm{Sub}(e(u), v(u))} \max(x, y) \\ &\quad \text{where } x = \mathit{step}(\mathit{generalize}(\forall u.\, \mathrm{K}_{w(u)}\, e(u))) \\ &\qquad\quad y = \mathit{step}(\mathit{generalize}(\forall u.\, \mathrm{K}_{w(u)}\, v(u))) \end{aligned}$$

It is straightforward to see that this is an optimization problem with optimal and overlapping substructure, which we can solve efficiently with memoization techniques.

With this compiling algorithm, we are now able to handle any chain access expressions. Furthermore, this algorithm optimizes the generated Pregel program in two aspects. First, this algorithm derives a message passing scheme with a minimum number of supersteps, thus reduces unnecessary cost for launching Pregel supersteps during execution. Second, by extending the memoization technique, we can ensure that a chain access expression will be evaluated exactly once even if it appears multiple times in a Palgol step, avoiding redundant message passing for the same value.

4.2.2 Compiling Palgol steps

Having introduced the compiling algorithm for remote data reads in Palgol, here we give a general picture of the compilation for a single Palgol step, as shown in Figure 4.7. The computational content of every Palgol step is compiled into a *main superstep*. Depending on whether there are remote reads and writes, there may be a number of *remote reading supersteps* before the main superstep, and a *remote updating superstep* after the main superstep.

We will use the main computation step of the SV program (lines 5–12 in Figure 4.3) as an illustrative example for explaining the compilation algorithm, which consists of the following four steps:

1. We first handle neighborhood access, which requires a sending superstep that provides all the remote data for the loops from the neighbors' perspective. This sending superstep is inserted as a remote reading superstep immediately before the main superstep.

2. We analyze the chain access expressions appearing in the Palgol step with the algorithm in Section 4.2.1, and corresponding remote reading supersteps are inserted in the front. (For the SV algorithm, the only interesting chain access expression is $D[D[u]]$, which induces two remote reading supersteps realizing a request-reply conversation.)

3. Having handled all remote reads, the main superstep receives all the values needed and proceeds with the local computation. Since the local computational

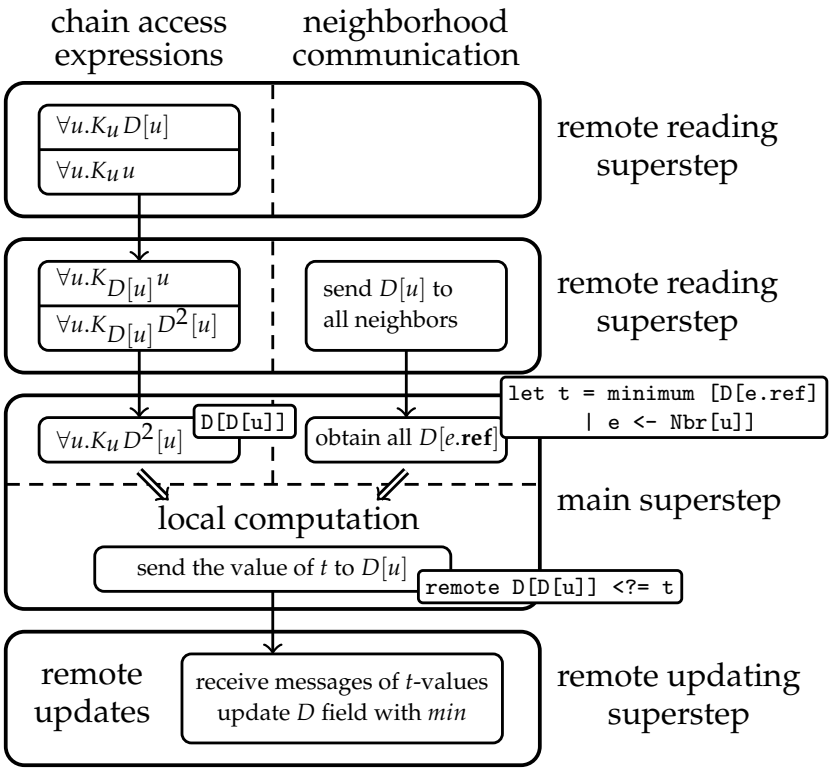


Figure 4.7: Compiling a Palgol step to Pregel supersteps.

content of a Palgol step is similar to an ordinary programming language, the transformation is straightforward.

4. What remain to be handled are the remote assignments, which require sending the updating values as messages to the target vertices in the main superstep. Then an additional remote updating superstep is added after the main superstep; this additional superstep reads these messages and updates each field using the corresponding remote updating operator.

4.2.3 Compiling sequences and iterations

Finally, we look at the compilation of sequence and iteration, which assemble Palgol steps into larger programs. A Pregel program generated from Palgol code is essentially a *state transition machine* (STM) combined with computation code for each state. Every Palgol step is translated into a "linear" STM consisting of a chain of states corresponding to the supersteps like those shown in Figure 4.7, and the compilation of

a Palgol program starts from turning the atomic Palgol steps into linear STMs, and implements the sequence and iteration semantics to construct more complex STMs.

Compilation of sequence: To compile the sequence, we first compile the two component programs into STMs, then a composite STM is constructed by simply adding a state transition from the end state of the first STM to the start state of the second STM.

Compilation of iteration: We first compile the loop body into an STM, which starts from some state S_{start} and ends in a state S_{end}, then we extend this STM to implement the fixed-point semantics. Here we describe a generalized approach which generates a new STM starting from state $S_{start'}$ and ending in state $S_{end'}$:

1. First, a check of the termination condition takes place right before the state S_{start}: if it holds, we immediately enters a new exit state $S_{exit'}$; otherwise we execute the body, after which we go back to the check by adding a state transition from S_{end} to S_{start}. This step actually implements a while loop.

2. The termination check is implemented by an OR aggregator to make sure that every vertex makes the same decision: basically, all vertices determine whether their local fields stabilize during a single iteration by storing the original values beforehand, and the aggregator combines the results and makes it available to all vertices.

3. We add a new start state $S_{start'}$ and make it directly transit to S_{start}. This state is for storing the original values of the fields, and also to make the termination check succeed in the first run, turning the while loop into a do-until loop.

Optimizations: In the compilation of sequence and iteration, two optimization techniques are used to reduce the number of states in the generated STMs and can remove unnecessary synchronizations. Due to space restrictions, we will not present all the details here, but these techniques share similar ideas with Green-Marl's "state merging" and "intra-loop state merging" optimizations [19]:

- *state merging*: whenever it is safe to do so, the Green-Marl compiler merges two consecutive states of vertex computation into one. In the compilation of sequence in Palgol, we can always safely merge the end state of the first STM

Table 4.1: Datasets for performance evaluation

Dataset	Type	Vertices	Edges	Description
Wikipedia	Directed	18,268,992	172,183,984	the hyperlink network of Wikipedia
Facebook	Undirected	59,216,214	185,044,032	a friendship network of the Facebook
USA	Weighted	23,947,347	58,333,344	the USA road network
Random	Chain	10,000,000	10,000,000	a chain with randomly generated values

and the start state of the second STM, resulting in a reduction of one state in the composite STM.

- *intra-loop state merging*: this optimization merges the first and last vertex-parallel states inside Green-Marl's loops. In Palgol, we can also discover such chance when iterating a linear STM inside a fixed-point iteration.

4.3 Evaluation of Palgol

In this section, we evaluate the overall performance of Palgol and the state-merging optimisations introduced in the previous section. We compile Palgol code to Pregel+[4], which is an open-source implementation of Pregel written in C++.[5] We have implemented the following six graph algorithms on Pregel+'s basic mode, which are the PageRank [1], Single-Source Shortest Path (SSSP) [1], Strongly Connected Components (SCC) [21], Shiloach-Vishkin Algorithm (SV) [21], List Ranking Algorithm (LR) [21] and Minimum Spanning Forest (MSF) [18]. Among these algorithms, SCC, SV, LR and MSF are non-trivial ones which contain multiple computing stages. Their Pregel+ implementations are included in our repository for interested readers.

Table 4.2: Comparison of execution time between Palgol and Pregel+ implementation

Dataset	Algorithm	4 nodes		8 nodes		12 nodes		16 nodes	
		Pregel+	Palgol	Pregel+	Palgol	Pregel+	Palgol	Pregel+	Palgol
	SSSP	8.33	10.80	4.47	5.61	3.18	3.83	2.41	2.85
Wikipedia	PageRank	153.40	152.36	83.94	82.58	61.82	61.24	48.36	47.66
	SCC	177.51	178.87	85.87	86.52	61.75	61.89	46.64	46.33
Facebook	SV	143.09	142.16	87.98	86.22	67.62	65.90	58.29	57.49
Random	LR	56.18	64.69	29.58	33.17	19.76	23.48	14.64	18.16
USA	MSF	78.80	82.57	43.21	45.98	29.47	31.07	22.84	24.29

4.3.1 Overhead of the DSL

In our performance evaluation, we use three real-world graph datasets (Facebook[6], Wikipedia[7], USA[8]) and one synthetic graph, and some detailed information is listed in Table 5.1. The experiment is conducted on an Amazon EC2 cluster with 16 nodes (whose instance type is m4.large), each containing 2 vCPUs and 8G memory. Each algorithm is run on the type of input graphs to which it is applicable (PageRank on directed graphs, for example) with 4 configurations, where the number of nodes changes from 4 to 16. We measure the execution time for each experiment, and all the results are averaged over three repeated experiments. The runtime results of our experiments are summarized in Table 4.2.

Remarkably, for most of these algorithms (PageRank, SCC, SV and MSF), we observed highly close execution time on the compiler-generated programs and the manually implemented programs, with the performance of the Palgol programs varying between a 2.53% speedup to a 6.42% slowdown.

For SSSP, we observed a slowdown up to 29.55%. The main reason is that the human-written code utilizes Pregel's *vote_to_halt()* API to deactivate converged vertices during computation; this accelerates the execution since the Pregel system skips invoking the *compute()* function for those inactive vertices, while in Palgol, we check the states of the vertices to decide whether to perform computation. Similarly,

[4] http://www.cse.cuhk.edu.hk/pregelplus

[5] Palgol does not target a specific Pregel-like system. Instead, by properly implementing different back ends of the compiler, Palgol can be transformed into any Pregel-like system, as long as the system supports the basic Pregel interfaces including message passing between arbitrary pairs of vertices and aggregators.

[6] https://archive.is/o/cdGrj/konect.uni-koblenz.de/networks/facebook-sg

[7] http://konect.uni-koblenz.de/networks/dbpedia-link

[8] http://www.dis.uniroma1.it/challenge9/download.shtml

Table 4.3: Comparison of the compiler-generated programs before/after optimization

Dataset	Algorithm	Number of Supersteps			Execution Time		
		Before	After	Comparison	Before	After	Comparison
Wikipedia	SSSP	147	50	−65.99%	5.36	2.85	−46.83%
	PageRank	93	32	−65.59%	45.57	47.66	4.58%
	SCC	3819	1278	−66.54%	106.03	46.33	−56.30%
Facebook	SV	31	23	−25.81%	52.37	57.49	9.78%
Random	LR	77	52	−32.47%	17.54	18.16	3.51%
USA	MSF	318	192	−39.62%	26.67	24.29	−8.95%

we observed a 24% slowdown for LR, since the human-written code deactivates all vertices after each superstep, and it turns out to work correctly. While voting to halt may look important to efficiency, we would argue against supporting voting to halt as is, since it makes programs impossible to compose: in general, an algorithm may contain multiple computation stages, and we need to control when to end a stage and enter the next; voting to halt, however, does not help with such stage transition, since it is designed to deactivate all vertices and end the whole computation right away.

4.3.2 Effectiveness of the fusion optimization

In this subsection, we evaluate the effectiveness of the "state merging" optimization mentioned in Section 4.2.3, by generating both the optimized and unoptimized versions of the code and executing them in the same configurations. We use all the six graph applications in the previous experiment, and fix the number of nodes to 16. The experiment results are shown in Table 4.3.

The numbers of supersteps in execution are significantly reduced, and this is due to the fact that the main iterations in these graph algorithms are properly optimized. For applications containing only a simple iteration like PageRank and SSSP, we reduce nearly 2/3 supersteps in execution, which is achieved by optimizing the three supersteps inside the iteration body into a single one. Similarly, for SCC, SV and LR, the improvement is around 2/3, 1/4 and 1/3 due to the reduction of one or two superstep in the main iteration(s). The MSF is a slightly complicated algorithm containing multiple stages, and we get an overall reduction of nearly 40% supersteps in execution.

While this optimization reduces the number of supersteps, and thus the number of global synchronizations, it does not necessarily reduce the overall execution time

since it incurs a small overhead for every loop. The optimization produces a tighter loop body by unconditionally sending at the end of each iteration the necessary messages for the next iteration; as a result, when exiting the loop, some redundant messages are emitted (although the correctness of the generated code is ensured). This optimization is effective when the cost of sending these redundant messages is cheaper than that of the eliminated global synchronizations. In our experiments, SSSP and SCC become twice as fast after optimization since they are not computationally intensive, and therefore the number of global synchronizations plays a more dominant role in execution time; this is not the case for the other algorithms though.

4.4 Problem of Palgol with the channel mechanism

Up to now, we have described Palgol, a domain-specific language to describe Pregel algorithms in high-level based on the idea of remote access. By using a rule-based compilation algorithm, a Palgol program can be executed on Pregel+'s basic mode, a standard implementation of the Pregel model, and the performance is comparable to the carefully optimized hand-written code. However, Pregel+ inherits Pregel's drawbacks that its monolithic message mechanism is incapable of dealing with multiple performance issues at the same time, and for complex graph algorithms like the SV algorithm, the execution time of a Pregel+ implementation can be 3.39× times slower than an implementation using our channel mechanism (see Section 3.3). A natural question here is that can we compile a Palgol program to an efficient program in our channel-based Pregel system? In the rest of this chapter, we discuss the challenges and introduce our solutions to this problem.

4.4.1 The main challenges

The channel mechanism is an extension of Pregel's message mechanism that allows several irregular communication patterns to be implemented in more efficient ways, and those special implementations are encapsulated into the optimized channels (presented in Table 3.2) and used by the programmers in their programs as plug-ins. There is no doubt that the Pregel-channel can simulate a standard Pregel system with its basic channels (in Table 3.1), and therefore a Palgol program can compile

to Pregel-channel without big change in its compilation algorithm. However, such transformation ignores the potential performance issues in the graph computation and cannot make use of the optimizations in Pregel-channels. The real question here is that, when compiling a Palgol program to Pregel-channel, can we choose the most suitable optimizations (channels) in its implementation to maximize the performance?

In the original Palgol compiler, it sees a graph computation as a bunch of Palgol steps connected by the combinators called sequence and iteration, and in each Palgol step it is basically a piece of vertex-centric computation using remote access to represent the communications between the vertices. More specifically, the Palgol compiler focuses on transforming various remote access primitives to bulk-synchronous message passing, which include the chain access (e.g., `P[P[u]]`), neighborhood access and remote writes. A natural idea to extend the Palgol compiler is to map each remote access primitives to every additional optimized channel, but unfortunately this straightforward approach is not viable due to the following reasons.

- **Cost estimation.** In Pregel-channel, due to the existence of various channels, it requires the compiler to be able to estimate the communication cost for every possible transformation, which is a huge burden in design and implementation. Moreover, in a Palgol program, we need to handle the composition of two or more remote access primitives, making the cost estimation more complicated.

- **Scalability.** The one-to-one mapping from every remote access primitive to every channel also limits the scalability of this approach since whenever we add a new optimized channel in Pregel-channel, we need to consider the transformation from every remote access primitive to that channel. It will significantly increase the complexity of the whole compilation procedure.

We believe that the original Palgol's compiler cannot be easily extended to support the Pregel-channel system. This requires us to think about a different approach that allows the compiler to properly understand the graph computation and can truly estimate the communication cost in the implementation.

4.4.2 A systematic solution using relational model

In this thesis, we design a SQL-style declarative language for describing graph computations and provide a novel compilation algorithm to compile Palgol to the Pregel-channel system. This language captures the vertex-centric graph computation using relational queries, and for each graph query, by enumerating all possible query plans (join orders and the type for each two-table join), we identify the plan that has the minimum communication cost by calculating how much data need to be moved in the query evaluation. In particular, the optimizations in Pregel-channel are regarded as special joins of two tables, and thus can be involved in the enumeration of query plans.

The technical contributions of our new language is summarized as follows:

- We propose a relational computation model for vertex-centric graph processing. By using an intuitive tabular representation for graphs, graph transformations are expressed as the inner join of a series of tables followed by an aggregation, which does not require users to explicitly specify the computation on each vertex as well as their interactions.

- We design a SQL-like language based on the new relational computation model, and implement it by reducing the problem of transforming the graph queries in our language to a Pregel program, to the problem of deciding the join order for hash-partitioned join algorithm [47]. We solve this problem by the dynamic programming technique with our cost model to minimize the communication cost for the query plan.

- We show how to obtain optimal vertex-centric programs by demonstrating that two useful optimizations [21, 28] can be easily detected in our high-level model, and the integration of such analysis can be achieved by a simple extension of our join-based compilation algorithm.

- We have fully implemented the compiler[9], and the experiment results convincingly show that our compilation algorithm achieves similar efficiency for many representative graph algorithms on large graph dataset. For the connected component problem, our framework even outperforms the state-of-the-art algorithm with a good margin.

[9]The source code of our system can be accessed at https://bitbucket.org/zyz915/sql-core.

4.5 SQL-core: a new compilation engine

In this section, we introduce our high-level model for graph computation, which includes a tabular representation for graphs, and a relational computation model for writing graph transformations as queries on the graph. It implicitly captures the vertex-centric feature, as will be seen later.

4.5.1 A tabular graph representation

In our model, graph has a directed adjacency structure and has user-defined attributes associated with each vertex and edge. Every vertex is assumed to have a unique identifier with integer type. Then, the graph data is represented by a collection of tables with the following schema:

- `VertexTable(id:Int, attr:a)` associates each vertex id with a vertex attribute having a user-defined type *a*. Users can define any number of vertex attributes in a graph computation.

- `EdgeTable(src:Int, dst:Int, attr:a)` stores the directed edges, each being a tuple containing the source vertex *src*, the destination vertex *dst* and an optional edge attribute *attr* with user-defined type *a*. For a graph computation, there is a unique and immutable table `Edge` having this schema[10], which defines the structure of the graph as well as the edge attributes.

- `GlobalValue(value:a)` contains a single value having a user-defined type *a* indicating a global value. Users can use any number of global values in a graph computation. We include this special table due to its usefulness in many graph computations.

We restrict the tables used in the graph computation to simplify the compilation, and it also helps us generate high performance code for our language.

[10] Supporting multiple edge tables is not technically difficult, but graph algorithms rarely require this feature.

4.5.2 A join-filtering-aggregation model

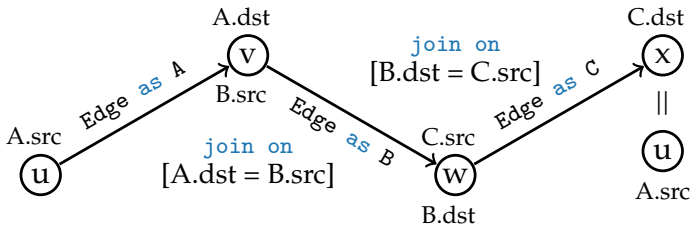
In our model, graph transformation is defined as a relational graph query over the tables having a restricted schema, and generates either a vertex table or a global value. The graph query can only use a subset of relational operations (select, join, group by, aggregation and simple predicates), and it also has additional restrictions in order to be compiled to an efficient vertex-centric program. Generally, a graph transformation in our model consists of the following three steps:

- **table join**: first, we calculate the *inner join* of a series of tables with our defined schema (except the global values) to represent a graph pattern, and we require that for every join, the two tables have a single attribute in common, which acts as the join attribute.
- **filtering**: then, having the join result, we filter the rows by predicates, which can only access the values of the current row and the global values, using arithmetic operations and comparison only.
- **aggregation**: finally, the join result is converted (by a select clause) to either a vertex table through a `group by` operation over a vertex id column, or a single global value through an aggregation function.

Here, we use triangle counting as an example to see how to write graph transformations in this model. Suppose the input is an edge table having the schema *Edge(src:Int, dst:Int)* without the edge attribute. The idea of using relational query for triangle counting is illustrated in Figure 4.8. We first enumerate all the distinct paths $u \to v \to w \to x$ through table join, and after obtaining all the tuples (u, v, w, x) connected by directed edges, we use the predicate $x = u$ to ensure that the path forms a triangle, and the predicates $u < v$ and $v < w$ are to eliminate the equivalent permutations. Finally, the aggregation function `count` in the `select` clause counts the number of rows, and the result is stored as a global value.

In comparison, we present a triangle counting program in the vertex-centric message passing model. For each vertex u, we use $Out(u)$ to represent u's outgoing adjacent list. Then the algorithm consists of the following three supersteps.

- **Step 1**: For each $v \in Out(u)$, vertex u sends its own vertex id to v if $u < v$;




```
select count(1)          -- aggregation
from Edge as A           -- table join
  inner join Edge as B on A.dst = B.src
  inner join Edge as C on B.src = C.dst
where A.src < B.src      -- filtering
  and B.src < C.src
  and A.src = C.dst
```

Figure 4.8: A graph query example of triangle counting using the standard SQL syntax.

- **Step 2**: Every vertex v receives its incoming neighbors' vertex id from the message list, which is stored as a new list $In(v)$. Then for each $u \in In(v)$ and $w \in Out(v)$, vertex v sends u to w if $v < w$;

- **Step 3**: Every vertex w receives a list of vertex ids, which are the vertices that can reach w in exactly two steps. Besides, each occurrence of u in this list indicates a distinct path from u to w. We store the list as $In^2(w)$. Then, for each $u \in Out(w)$, vertex w counts the occurrence of u in $In^2(w)$, and the sum is the number of triangles having w as the largest vertex id.

The vertex-centric triangle counting algorithm is more complicated due to the explicit message passing in the algorithm description. Also, we use u, v, w to distinguish the role of the current vertex in each step to help readers understand the intended interactions. However, this vertex-centric program and the SQL query in Figure 4.8 has a close relationship, and using our language, we can derive this vertex-centric implementation from the graph query.

4.5.3 An overview of SQL-core

In this part, we present our language for writing graph computations in our proposed model. This language is in general similar to SQL, but its syntax is designed to reflect those restrictions we made on the computation model. It is also a compositional

language that allows users to easily perform the iteration over any kind of graph transformations.

The syntax of the language is defined in Figure 4.9. As described by the syntactic category *trans*, a graph transformation contains a relational query following our high-level model, and the result table is bound to a variable. Then, such graph transformations can be further composed or iterated as shown in the syntactic category *prog*. There are two kinds of iterations in our language, the *for loop* and the *fixed-point iteration*. The former one iterates a program by a fixed number of rounds, and the latter one iterates a program until the specified table stabilizes.

The syntactic category *query* defines the relational query in our language, which includes Q_v for generating a vertex table and Q_g for producing a global value. Their differences are reflected in the `select` and `group by` clauses, while the table join and the filtering are the same.

The syntax of table join is special in our language to better reflect the restrictions in our computation model. We present two query examples in Figure 4.10, which are part of the implementation of PageRank and triangle counting. We first list all the vertices and their attributes used in a graph computation, then we assign a unique variable to each vertex and vertex attribute and connect then via link. Each link is essentially a table specifying the relation between two variables. In our language, we just put the tables after the keyword `from` to describe such graph computation.

A valid join then requires the variables and the links to form a spanning tree. Remember that in our computation model, we require that for each join the two tables must have exactly one attribute in common. Here, two links can join if they share an endpoint, and the join result is a connected subgraph. Similarly, two connected subgraphs (intermediate join results) containing disjoint set of links can join if they share a node. In both cases, the joint node automatically acts as the join attribute, so there is no need to explicitly specify the join attributes in our language. We also note that, in our language, the order of tables does not matter, since deciding the join order is completely delegated to the compiler (details in Section 4.6.3).

The `where` clause contains the predicates to filter out the unwanted tuples in the join result. Predicates are arithmetic expressions and comparison using only the columns in the join result, and we do not allow a predicate to contain any sub-query. Finally, the `select` clause generates either a vertex table or a global value. We present

$$
\begin{array}{lcl}
\mathit{prog} & ::= & \mathit{trans} \mid \mathit{prog}_1, \ldots, \mathit{prog}_n \mid \mathit{iter} \\
\mathit{trans} & ::= & \textbf{set } \mathit{name} : \mathit{sig} \ = \ \mathit{query} \\
\mathit{sig} & ::= & \textbf{vertex\_table } (\mathit{type}) \mid \textbf{global\_value } (\mathit{type}) \\
\mathit{type} & ::= & \textbf{vid} \mid \textbf{int} \mid \textbf{float} \\
\mathit{query} & ::= & Q_v \mid Q_g \\
\mathit{iter} & ::= & \textbf{do } \mathit{prog} \textbf{ until fix } (\mathit{name}) \\
 & \mid & \textbf{for } \mathit{var} \textbf{ in } \mathit{num} \textbf{ to } \mathit{num} \textbf{ do } \mathit{prog} \textbf{ end} \\
Q_v & ::= & \textbf{select } \mathit{var}, \ \mathit{column} \\
 & & \textbf{from } \mathit{table}_1 \textbf{ join } \ldots \textbf{ join } \mathit{table}_m \\
 & & \textbf{where } \mathit{expr}_1 \textbf{ and } \ldots \textbf{ and } \mathit{expr}_r \\
 & & \textbf{group by } \mathit{var} \\
Q_g & ::= & \textbf{select } \mathit{func} \ (\mathit{expr}) \\
 & & \textbf{from } \mathit{table}_1 \textbf{ join } \ldots \textbf{ join } \mathit{table}_m \\
 & & \textbf{where } \mathit{expr}_1 \textbf{ and } \ldots \textbf{ and } \mathit{expr}_r \\
\mathit{table} & ::= & \mathit{name} \ (\mathit{var}_1 \ldots \mathit{var}_s) \\
\mathit{column} & ::= & \mathit{expr} \mid \mathit{func} \ (\mathit{expr}) \\
\mathit{expr} & ::= & \mathit{num} \mid \mathit{var} \mid \mathit{expr} \ \mathit{op}_b \ \mathit{expr} \mid \mathit{op}_u \ \mathit{expr} \\
 & \mid & \textbf{if } (\mathit{expr}) \textbf{ then } \mathit{expr} \textbf{ else } \mathit{expr} \\
\mathit{func} & ::= & \textbf{max} \mid \textbf{min} \mid \textbf{count} \mid \textbf{sum} \mid \textbf{avg}
\end{array}
$$

Figure 4.9: The essential syntax of the query language.

both examples for Q_v and Q_g in Figure 4.10.

4.5.4 Case studies

In Figure 4.11 we implement the PageRank and single source shortest path (SSSP) using our language. They are in general similar, so we just take PageRank as an example. A PageRank computation takes a fixed number of iterations, and in each round the vertex table `Pr(u,pr)` stores each vertex's tentative pagerank and is updated based on the previously computed result. The other tables involved are the edge table `Edge(u,v)`, a vertex table `Degree(u,deg)` storing each vertex u's out-degree, and a global value `Size(g)` storing the number of vertices in the graph. The new pagerank for each vertex v is calculated from all the tuples (u, v, pr, deg, g) having the same v, by summing up the pr/deg (to generate an intermediate table `Msg` in line 3) and then performing a local computation (in line 7 by a separate query) on each vertex.

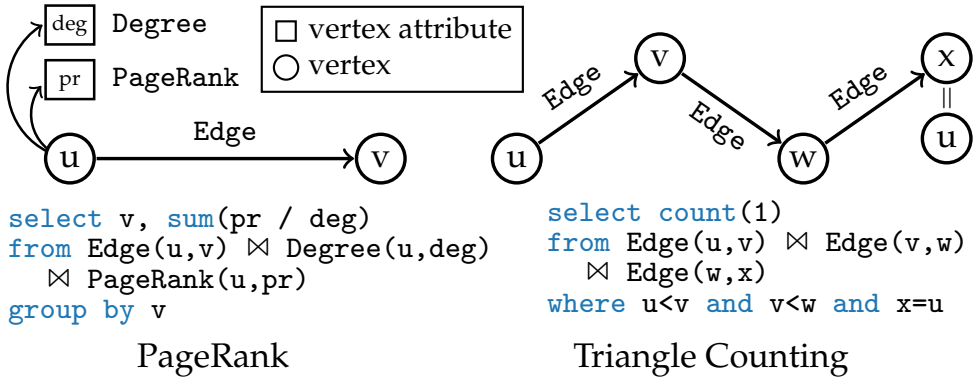


Figure 4.10: Two query examples in our language.

PageRank

```
for i in 1 to 30 do
  set Msg : vertex_table(float) =
    select v, sum(pr / deg)
    from Edge(u, v) join Pr(u, pr) join Deg(u, deg)
    group by v
  set Pr : vertex_table(float) =
    select u, 0.85 * msg + 0.15 / g
    from Msg(u, msg) join Size(g)
end
```

Single-Source Shortest Path

```
do
  set Msg : vertex_table(int) =
    select v, min(du + len)
    from Edge(u, v, len) join Dist(u, du) join
         Dist(v, d)
    group by v
  set Dist : vertex_table(int) =
    select u, min(old, new)
    from Dist(u, old) join Msg(u, new)
until fix(Dist)
```

Figure 4.11: PageRank and single-source shortest path (SSSP) implemented in our language. The tables are assumed to be initialized.

4.6 Compiling SQL-core to Pregel

In this section, we introduce our cost-based join algorithm for transforming a graph query to a query plan, which is the key step in our compilation. The algorithm is generally based on the multiprocessor hash-based join algorithm [47], but in the domain of graph processing, the following facts make our algorithm different from the previous work:

- Usually in graph computations, the join does not produce an exceedingly large intermediate table, making the memory consumption less a problem. Instead, we are more concerned with the communication cost.

- In distributed-memory graph processing, optimizations play an important role in achieving high-performance, therefore a straightforward join-based query evaluation may suffer various kinds of performance issues. This paper also focuses on extending the hash-join algorithm to make use of those optimizations.

We present the compilation algorithm in two parts. The basic hash-join algorithm and our special cost model for communication cost are presented in this section, and the extension of our compilation algorithm for supporting various optimizations will be discussed in the next section. The whole compilation also includes other steps like parsing and code generation, but they are either classic or uninteresting, so we can safely skip them in this paper.

4.6.1 Distribution of tables

Processes are the basic computation unit in our graph system. They share no memory, but can communicate with each other though the Message-Passing Interface (MPI).

Tables are horizontally partitioned across the processes in the system, and the partitioning is decided by a hash function and a column of the table. The hash function maps a vertex id to a process id, then for each tuple, the hash value of the chosen column decides in which process it is stored.

The repartitioning of a table is a basic operation in our framework, which changes the hash column to another one. When triggered, all the processes iterate over the

tuples in the table stored on its local memory, and send each tuple to the designated process decided by the new column.

This table distribution strategy plays a central role in our compilation algorithm, which is the key to mapping our language to the vertex-centric execution model. We justify our choice as follows:

- We can easily apply the multiprocessor version of the hash-based join algorithms [47] to evaluate the query in parallel. In particular, if both tables are partitioned by the join attribute, then all the tuples with the same join attribute (which is called a *bucket*) will be stored on the same process, making the join operation efficient.
- The hash-based partitioning strategy is similar to Pregel's design choice, which assigns vertices to machines based on a random hash. Having this connection, we manage to implement the query evaluation on Pregel, which avoids redundant work.
- The hash-based partitioning does not provide the best load balancing. However, it does not require any preprocessing, which is important to us since repartitioning may be frequently used in the query evaluation, especially for complex joins.
- There are already lots of optimizations developed for Pregel systems, and it is also our goal to automatically apply these optimizations in the query evaluation.

4.6.2 Inner join using vertex-centric computation

First, let us consider the join of two tables $T_1(a, b)$ and $T_2(a, c)$ having a as the join attribute, which is the simplest case of join. The communication cost of the join depends on the partitioning of the two tables:

- If both tables are partitioned by the join attribute a, then it is obvious that all the tuples with the same join attribute a in both tables are on the same process. In this case, each bucket locally performs the join and introduces no communication cost.
- Otherwise, suppose $T_1(a, b)$ is partitioned by b, then we perform a repartitioning on column a by sending each tuple (a, b) in T_1 to process $hash(a)$ through

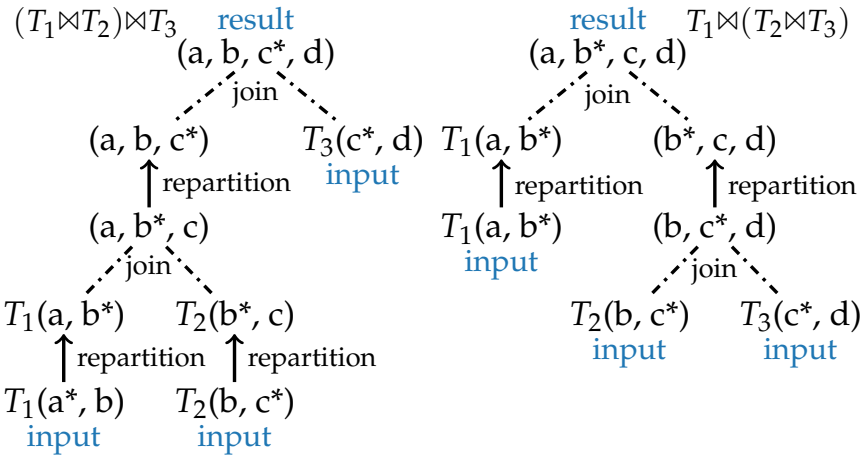


Figure 4.12: Two different join orders to evaluate $T_1 \bowtie T_2 \bowtie T_3$.

message passing. We do the same thing for T_2 if it is not partitioned by a. Then we perform the local join.

To clearly describe the compilation algorithm, it is helpful to syntactically distinguish a table's different ways of partitioning. We introduce the following notation $T_1(a*, b)$ with an asterisk attached to one of the columns, representing the concrete data distribution of T_1 that is partitioned by column a. Similarly, $T_1(a, b*)$ is T_1 partitioned by column b. This notation also applies to the intermediate result of a join. For example, we can say that the join result of $T_1(a*, b)$ and $T_2(a*, c)$ is a table $(a*, b, c)$. Here, the order of the columns does not matter. The join strategy described above is our *default* strategy.

4.6.3 Efficient join ordering

When joining more than two tables, different join orders may result in different communication cost, and the decision of ordering is further affected by the subsequent computation. To see this, consider the following example that calculates the join of $T_1(a*, b)$, $T_2(b, c*)$ and $T_3(c*, d)$. Two different query plans are presented in Figure 4.12, where the left one joins T_1 and T_2 first, and the right one joins T_2 and T_3 first. It should be noted that joining $T_1(a*, b)$ and $T_3(b, c*)$ is far too costly due to the absence of a common attribute, so we consider it as an invalid join.

We can see that the two query plans are likely to have different communication cost, since they use different number of repartition operations. However, yet we cannot say that the second query plan is better than the first one or vice versa. We should note that $(a, b, c*, d)$ and $(a, b*, c, d)$ are the same table partitioned by different columns. Suppose the next join is on column c, then the second plan requires an additional repartitioning operation, which is not reflected in the current plan's cost. Therefore, just considering the cost of table generation can lead to the sub-optimal problem.

To calculate the best join order in the presence of subsequent computation, the join algorithm should know in advance what column the solution is partitioned by. Therefore, the input of the algorithm should is a pair (S, x) where S contains the tables to join, and x is the column that the result table is partitioned by. Here, x should be an attribute of some $T \in S$, otherwise the input is not valid. For example in Figure 4.12, we say that the query plan on the left is a solution for $(\{T_1, T_2, T_3\}, c)$, and the query plan on the right is a solution for $(\{T_1, T_2, T_3\}, b)$.

Given a query, let S be the tables in the `from` clause and xs be the columns appeared in S. For every $x \in xs$, we find the best query plan for the input (S, x) using Algorithm 4.1. This is a top-down description of the hash-join algorithm, in which we solve a problem (S, x) by breaking it into smaller sub-problems with the same structure. It basically enumerates all possible query plans and selects the one with the minimum cost. We leave the detailed explanation of the *cost()* function in the next subsection. To avoid redundant computation caused by solving the same problem repeatedly, we implement this algorithm in a bottom-up way using dynamic programming, ensuring that each (S, x) is evaluated exactly once.

We made the algorithm extensible by predefining the join strategies in a global array in line 2. The *default* strategy is implemented in line 18, which is exactly the join algorithm introduced in Section 4.6.2. Then, Algorithm 4.1 with the default join strategy is sufficient to compile any table join to a valid query plan and then to a vertex-program, but yet we cannot ensure the high efficiency. We will discuss the other join strategies in Section 4.7.3 for achieving better performance.

Table 4.4: The estimation of the communication cost and the table size for each type of node in the query plan.

node t	$cost(t)$	$size(t)$
`Leaf`(t_1)	0	n or m
`Repar`(t_1)	$size(t_1) * (w-1)/w$	$size(t_1)$
`Join`(t_1, t_2)	$cost(t_1) + cost(t_2)$	$size(t_1) * size(t_2)/n$
`ReqResp`(t_1, t_2)	$n * (1 - e^{-size(t_1)/n}) +$ $cost(t_1) + cost(t_2)$	$size(t_1)$

4.6.4 Cost estimation

In this part, we present the implementation of *cost()* to complete the Algorithm 4.1. The query plan has a tree structure where each leaf node is an input table and an internal node represents a result table of a repartitioning or a join (see Figure 4.12 as an example). Our *cost()* function applies to a node in the query plan and returns the communication cost to generate the result table. We also need an auxiliary function *size()* defined in the same way returning the estimated table size.

Three parameters n, m and w are used by our *cost()* and *size()* function, which represent the number of vertices and edges in the input graph and the number of processes launched by the user. One can also use the density (m/n) to implement these functions, and w can be safely replaced by a large enough value. The two functions are defined in Table 4.4 for each type of node in the query plan. To obtain a more precise communication cost, we also keep track of the record size of each table (see Section 4.7.1), but it is orthogonal to the implementation of these two functions.

4.6.5 Result table generation

Next, we briefly go through the result table generation, which transforms the join result to either a vertex table or a global value depending on the syntax (see Q_v or Q_g in Figure 4.9).

Vertex Table

A vertex table is generated from the join result via the *group by* operation over one of the columns in the table. The vertex attribute is calculated from an aggregate

function. The following query from PageRank shows a typical example of vertex table generation:

```
1 select v, sum(pr / deg)
2 from Edge(u, v) join Pr(u, pr) join Deg(u, deg)
3 group by v
```

The vertex table having v as the first column is generated from the join result $(u, v*, deg, pr)$. For each row, we first compute the expression inside the aggregation function (which is pr/deg in this example), then all the rows with the same attribute v sum up their values of pr/deg, generating a single value $sum(pr/deg)$ associated to vertex v. Those v's that have not appeared in the join result are assigned with the default value 0.

Global Value

The generation of a global value from the join result is simply implemented by all the processes performing an *all-reduce* operation using MPI. In such case, the actual data distribution of the join result does not matter, therefore we enumerate every column c and choose the query plan (S, c) having the minimum communication cost.

4.7 Deep optimizations for SQL-core

In this section, we present various strategies for optimizing the query plan for reducing the computation and communication cost. Our language not only applies several standard optimizations studied in relational database, but also makes use of graph-specific optimizations in the graph system. We show that our relational model provides abundant hints for the compiler to optimize different kinds of graph transformations.

4.7.1 Filtering the table entries

The query filters perform a straightforward task — filtering the rows and columns of a table — to reduce the size of intermediate tables as well as the communication cost. The technique is intuitive and standard: rows are eliminated based on the predicates, and columns are removed when they are not used in the consequent computation. We

push down the filters as early as possible. We also extend our cost-based join with an additional attribute, the record size, to estimate the communication more precisely.

4.7.2 Improved vertex-table generation

In Section 4.6.5, the generation of a vertex table consists of two steps: (1) obtain the join result partitioned by the vertex table's first column, and (2) perform a local *group by* on that column. In many cases, for example calculating the `Msg` table for PageRank and SSSP in Figure 4.11, the query plan in the first step ends with a repartitioning operation. Repartitioning is a relatively communication-intensive operation, since most rows in the join result are involved. However, when the next operation is *group by*, we can reduce the communication cost by shrinking the table before partitioning.

Our improved vertex-table generation consists of the following three steps: (1) obtain the join result partitioned by some column c; (2) each process groups the tuples on its local memory by column c and then repartition the join result by the vertex table's first column v, and (3) perform a second *group by* operation on v. It generates the same table due to the associativity of the aggregation function. In the new algorithm, we need to enumerate c and select the one that minimizes the total communication cost. Algorithm 4.1 estimates the cost of the first step, and the cost of the second step can be estimated by $n(w-1)(1-(1-1/w)^{s/n})$ by assuming a uniform degree distribution [48], where s is the size of the join result.

4.7.3 Introduction of the optimized channels

In the last part, extend our compiling algorithm to support optimization channels.

The request-respond channel

Several graph algorithms (like Shiloach-Vishkin [16] and LACC [31]) may use an auxiliary structure called *pointer graph* to keep track of the connectivity information during the computation. The pointer graph is essentially a forest of rooted trees consisting of all the vertices in the input graph. To store this structure, each vertex just maintains a field indicating its current parent vertex in the pointer graph (a root vertex points to itself), and communication occurs between a vertex with its parent or

children. A common operation on this data structure is to find the grandparent for every vertex, which is known to have skewed communication pattern.

We describe this operation in our language. The pointer graph can be represented by a vertex table `P(id, parent)`, and each vertex's grandparent is also a vertex table `GP(id, grandparent)`. The following query generates the grandparent table from the parent table:

```
1   select u, w
2   from P(u, v) join P(v, w)
3   group by u
```

The default join strategy converts $P(u*, v)$ to $P(u, v*)$ and performs a local join on column v, obtaining $(u, v*, w)$. Then, another repartitioning is required to generate the result table $(u*, w)$. In a vertex-centric view, each vertex u sends a message to its parent v in the first repartitioning, and every parent vertex v replies to its children the w in the second one. They are exactly the *request* and *respond* steps.

In practice, there are high-degree parent vertices v due to the algorithm logic, causing an imbalanced distribution for $P(u, v*)$. One solution proposed for this issue is the request-respond paradigm [21]. After all the vertices emit the requests, the processes will first merge the requests to the same parent vertex and then send out only the distinct requests. As a result, a high-degree parent can receive at most one request from each process instead of one request from each of its children, which effectively avoid the hot spot.

The detection of this optimization is achieved by extending Algorithm 4.1 with a new join strategy *reqresp* and adding a new `ReqResp`(t_1, t_2) node in the query plan. An auxiliary function *is_primary_key()* is used to decide whether a column in the table is the primary key.

The detailed implementation of this strategy is in Algorithm 4.2, and the communication cost is presented in Table 4.4 under the assumption that attribute u and v are independent. Note that the cost function for default join does not tell anything about the load balancing, but `ReqResp` plan is generally preferable when it is applicable.

The scatter-combine channel

Scalable graph computations are usually iterative, and it is a natural idea to take out repeated computation from the iteration. Let us consider the following example, which

is an iterative program that calculates the join of a vertex table $Value(v, w)$ and an edge table $Edge(u, v)$ in every iteration.

```
1 do
2   set T : vertex_table(int) =
3     select u, min(w)
4     from Edge(u, v) join Value(v, w)
5     group by u
6   ...
7 until fix(..)
```

At first glance, it is a request-respond pattern where each u sends a request to every neighbor and calculates the minimum response value. However, the real issue is not the load balancing but the heavy computation inside the iteration. In this query, the edge table $Edge(u, v)$ is immutable and there is no filter, meaning that we will send the vertex v's current value w to every neighbor of u in every iteration. A graph computation that frequently sends vertex attributes along a fixed set of edges is said to have the static messaging pattern [28]. A conventional implementation incurs heavy computational cost due to the sorting of the large message list, while the solution they proposed is organizing the edges in a particular order, so that the message passing can be implemented in a linear scan.

The requirements of this optimization is stricter than the request-respond pattern. The query should (1) be inside an iteration, (2) have the edge table `Edge`(u, v) in the join, (3) have no filter applicable to the current join, and (4) have u as the first column and some value associated on v as the second column, or vice versa. Each step can be easily detected by an auxiliary function, and due to the page limit, we do not present the detailed implementation in this paper.

4.8 Evaluation of SQL-core

In this section, we evaluate the performance of our generated code, and also compares our system with other frameworks. We select four representative algorithms in our evaluation, including PageRank (PR) [51, 1], Shiloach-Vishkin algorithm (SV) [16, 21], Triangle Counting (TC) and Single-Source Shortest Path (SSSP) [1]. The experiments are conducted on Amazon EC2 cluster of 16 nodes (with instance type r4.2xlarge), each having 8 vGPUs and 61 memory size. We use 128 processes (single-threaded) in our evaluation, and the datasets are listed in Table 5.1 using real-world graphs.

Table 4.5: Graph datasets used to evaluate our framework.

Graph	Vertices	Edges	Density	Description
DBpedia	18.27M	136.54M	7.47	The DBpedia hyperlink graph [49]
Queen_4147	4.15M	166.82M	40.23	3D structural problem [50]
HV15R	2.02M	283.07M	140.33	Computational Fluid Dynamics Problem [50]
uk-2005	39.45M	936.36M	23.73	2005 web crawl of .uk domain [50]
twitter7	41.65M	1.47B	35.25	twitter follower network [50]
sk-2005	50.64M	1.95B	38.50	2005 web crawl of .sk domain [50]

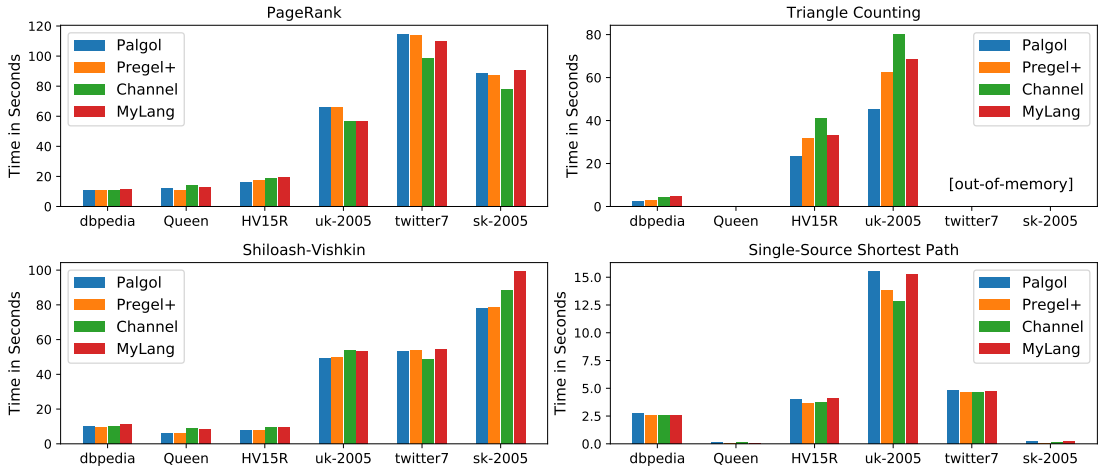


Figure 4.13: Performance of the unoptimized vertex-centric programs in various systems.

4.8.1 Performance without optimizations

We first evaluate the performance of our generated program using only Pregel's standard message passing interface. The main competitors are listed below:

- **Palgol**: a vertex-centric DSL on top of Pregel+ [21].
- **Pregel+**: the Pregel+ framework with the implementations of various Pregel algorithms by human.
- **SQL-core**: our DSL that compiles to the Pregel-channel system (see Chapter 3).
- **Channel**: the original Pregel-channel framework with two additional optimizations (introduced in Section 4.7.3 but not used in the current experiment).

All of these systems are implemented in C++ as a group of headers. There are also other DSLs for Pregel implemented on other systems, like Green-Marl [19] that generates C++ code and Fregel [20] with the Giraph [52] back end. We choose Palgol due to its expressive power for practical Pregel algorithm and its concrete implementation for efficiency.

The results are presented in Figure 4.13. On all the six graphs, we observed similar runtime for the four programs. Our language did generate the plan with the least message size, which makes our program comparable with human written code (Pregel+ and Channel). The Palgol language also generates the code with optimal messages size. These systems have subtle differences in implementation, like the choice of MPI function, serialization methods and so on, making the runtime differs a bit.

This experiment mainly reveals the performance characteristics of the back end system. In fact, our language is possible to be compiled to any of these back ends, but we choose the current system mainly due to the powerful optimizations it supports, which will be presented in the next subsection.

4.8.2 Performance with detected optimization

In this section, we focus on the SV and PageRank algorithm, which requires the optimization techniques presented in Section 4.7.3 to achieve high efficiency. The evaluation results are presented in Figure 4.14.

The SV algorithm uses both optimizations and is presented in the upper-part, including the unoptimized SV using our default join strategy (leftmost), and the optimized SV with both optimization techniques detected (rightmost). The original channel-based Pregel system also has these two optimizations, so we include it in the figure. We further include the linear algebra connected component (LACC) [31] algorithm here, the state-of-the-art scalable algorithm for finding connected components.

The compiler successfully detects the two optimizations. Compared to our unoptimized version, our optimized SV gains a huge speedup from 1.14× to 14.29× on all the six datasets, and compared to the original *Channel* implementation, this number is on average 2.7× (max 4.44×). For PageRank, we also observe a significant performance gain compared to unoptimized version (avg. 5.21×, max 7.45×), and it is faster than the PageRank in the original *Channel* system.

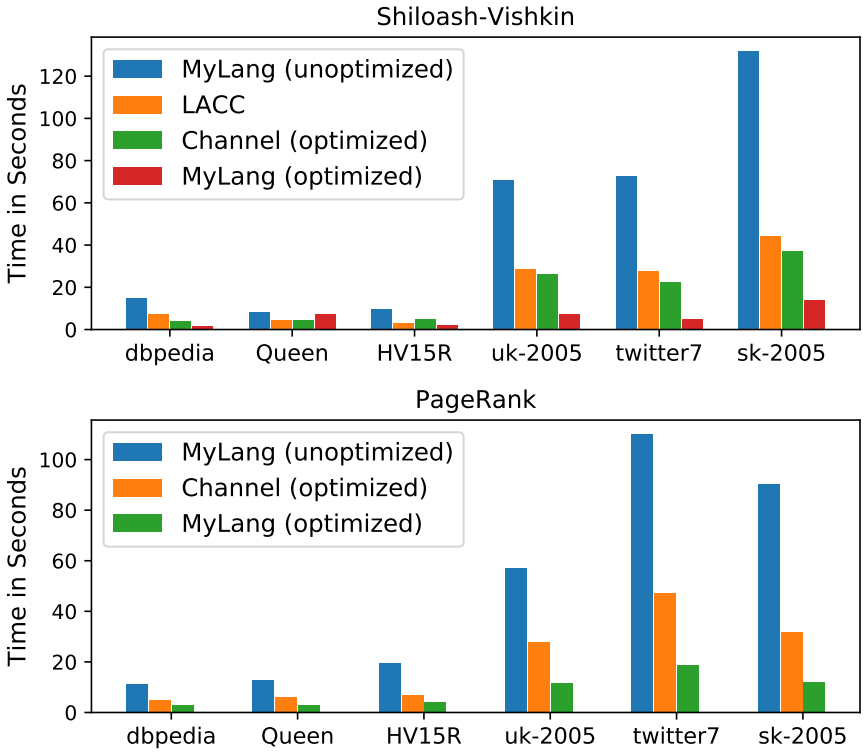


Figure 4.14: Performance of the Shiloach-Vishikin and PageRank algorithm in various systems. Both algorithm requires the optimization techniques introduced in Section 4.7.3.

4.9 Related work

Google's Pregel [1] proposed the vertex-centric computing paradigm, which allows programmers to think naturally like a vertex when designing distributed graph algorithms. Some graph-centric (or block-centric) systems like Giraph+[24] and Blogel [26] extends Pregel's vertex-centric approach by making the partitioning mechanism open to programmers, but it is still unclear how to optimize general vertex-centric algorithms (especially those complicated ones containing non-trivial communication patterns) using such extension.

Domain-Specific Languages (DSLs) are a well-known mechanism for describing solutions in specialized domains. To ease Pregel programming, many DSLs have been proposed, such as Palovca [53], s6raph [54], Fregel [20] and Green-Marl [19]. We briefly introduce each of them below.

Palovca [53] exposes the Pregel APIs in Haskell using a monad, and a vertex-centric program is written in a low-level way like in typical Pregel systems. Since this language is still low-level, programmers are faced with the same challenges in Pregel programming, mainly having to tackle all low-level details.

At the other extreme, the s6raph system [54] is a special graph processing framework with a functional interface, which models a particular type of graph algorithms containing a single iterative computation (such as PageRank and Shortest Path) by six programmer-specified functions. However, many practical Pregel algorithms are far more complicated.

A more comparable and (in fact) closely related piece of work is Fregel [20], which is a functional DSL for declarative programming on big graphs. In Fregel, a vertex-centric computation is represented by a pure step function that takes a graph as input and produces a new vertex state; such functions can then be composed using a set of predefined higher-order functions to implement a complete graph algorithm. Palgol borrows this idea in the language design by letting programmers write atomic vertex-centric computations called Palgol steps, and put them together using two combinators, namely sequence and iteration. Compared with Fregel, the main strength of Palgol is in its remote access capabilities:

- a Palgol step consists of local computation and remote updating phases, whereas a Fregel step function can be thought of as only describing local computation,

lacking the ability to modify other vertices' states;

- even when considering local computation only, Palgol has highly declarative *field access expressions* to express remote reading of arbitrary vertices, whereas Fregel allows only neighboring access.

These two features are however essential for implementing the examples in Section 4.5.4, especially the SV algorithm. Moreover, when implementing the same graph algorithm, the execution time of Fregel is around an order of magnitude slower than human written code; Palgol shows that Fregel's combinator-based design can in fact achieve efficiency comparable to hand-written code.

Another comparable DSL is Green-Marl [55], which lets programmers describe graph algorithms in a higher-level imperative language. This language is initially proposed for graph processing on the shared-memory model, and a "Pregel-canonical" subset of its programs can be compiled to Pregel. Since it does not have a Pregel-specific language design, programmers may easily get compilation errors if they are not familiar with the implementation of the compiler. In contrast, Palgol (and Fregel) programs are by construction vertex-centric and distinguish the current and previous states for the vertices, and thus have a closer correspondence with the Pregel model. For remote reads, Green-Marl only supports neighboring access, so it suffers the same problem as Fregel where programmers cannot fetch data from an arbitrary vertex. While it supports graph traversal skeletons like BFS and DFS, these traversals can be encoded as neighborhood access with modest effort, so it actually has the same expressiveness as Fregel in terms of remote reading. Green-Marl supports remote writing, but according to our experience, it is quite restricted, and at least cannot be used inside a loop iterating over a neighbor list, and thus is less expressive than Palgol.

Algorithm 4.1 The cost-based join algorithm. **Input:** a set of tables S and a column x by which the result is partitioned. **Output:** a query plan with the minimum cost.

```
1:  ▷ Predefined join strategies
2:  strategies ⇐ {default, reqresp, static }
3:  procedure SOLVE(S, x)
4:      ▷ Case 1: Dealing with one table.
5:      if |S|=1 then
6:          T ⇐ the only table in S
7:          return repartition_if_needed(Leaf(T), x)
8:      end if
9:      ▷ Case 2: Joining two tables.
10:     ret ⇐ null
11:     for (S_l, S_r) ← valid_split(S) do
12:         for s ← strategies do
13:             p ⇐ repartition_if_needed(JOIN(s, S_l, S_r), x)
14:             ▷ Cost estimation by cost()
15:             if ret = null or cost(p) < cost(ret) then
16:                 ret ⇐ p
17:             end if
18:         end for
19:     end for
20:     return ret
21: end procedure
22: ▷ The default join operation on two tables
23: procedure JOIN(default, S_l, S_r)
24:     c ⇐ common_attribute(S_l, S_r)
25:     p_l ⇐ SOLVE(S_l, c)
26:     p_r ⇐ SOLVE(S_r, c)
27:     return Join(p_l, p_r, c)
28: end procedure
```

Algorithm 4.2 The *reqresp* join strategy. **Input:** two sub-queries S_1 and S_2 to join. **Output:** a request-respond query plan or *null* if not applicable.

```
▷ The reqresp join operation on two tables
procedure JOIN(reqresp, S_l, S_r)
    c ⇐ common_attribute(S_l, S_r)
    if is_primary_key(S_r, c) then
        for u ← all_attributes(S) do
            p_l ⇐ SOLVE(S_l, u)
            p_r ⇐ SOLVE(S_r, c)
            return ReqResp(p_l, p_r, u)
        end for
    end if
    return null
end procedure
```

5

The Linear Algebra Approach

The linear algebra approach is yet another programming model for large-scale graph processing and has gained big success in some graph problems like the breadth-first search [56]. However, this approach is currently less popular than the vertex-centric paradigm, which is probably due to the more abstract way of thinking required in the linear algebraic programming of graph algorithms. This work tries to convince the readers that the linear algebra approach is actually very promising which can not only represent complex graph algorithms in an elegant way but also implement some graph computation more efficiently than the vertex-centric paradigm.

In this chapter, we propose two linear algebra algorithms, the FastSV algorithm for finding connected components in undirected graphs, and a linear algebra implementation of Boruvka's algorithm for finding the minimum spanning forests in undirected weighted graphs. Both algorithms are classic graph algorithms that have numerous applications in modern areas like biological data analysis [57, 58], cancer detection [59, 60], computer vision [61], clustering algorithms [62, 63], and scientific computing. Our work not only enriches the set of problems the linear algebra approach

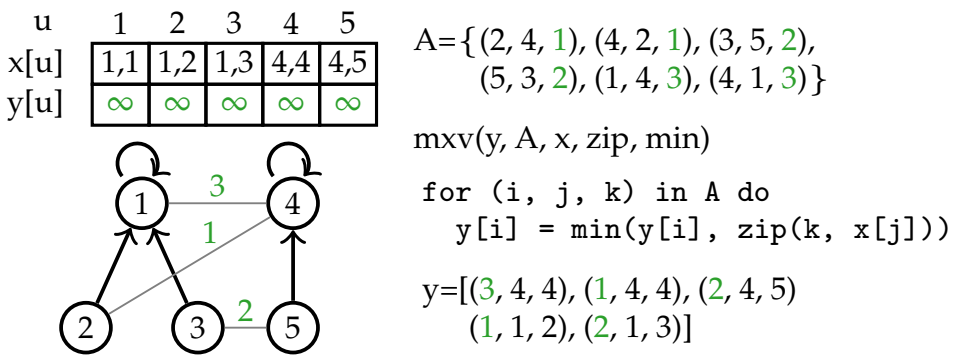


Figure 5.1: Illustration of the matrix-vector multiplication for min-edge picking. In Boruvka's algorithm, matrix **A** is an undirected weighted graph and vector x encodes both the supervertex id and the vertex's own id. The *mxv* finds for each vertex u the minimum tuple $(w, f[v], v)$ among the v's in u's adjacency list.

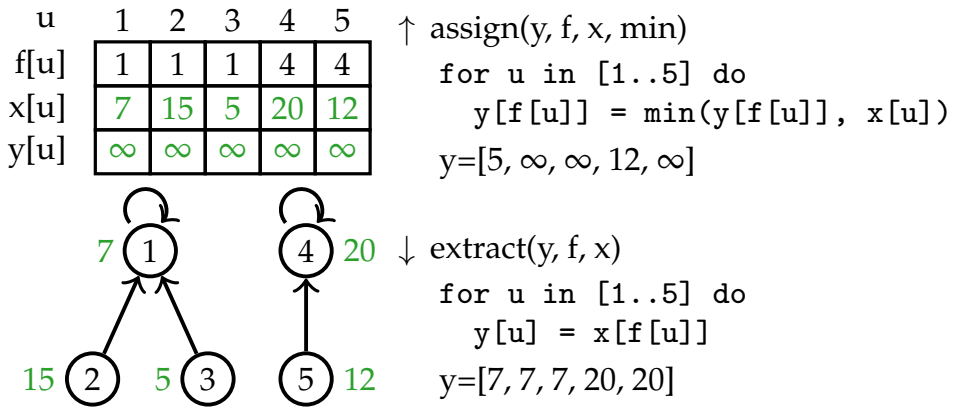


Figure 5.2: Illustration of the *assign* and *extract* operations (without a mask). In Boruvka's algorithm, vector f stores the supervertex of every vertex, which defines a flattened tree structure for each connected component. The *assign* operation let every supervertex collect the minimum value in the whole tree, and the *extract* operation let every vertex see the value on the root.

Algorithm 5.1 The SV algorithm. **Input:** An undirected graph $G(V, E)$. **Output:** The parent vector f.

```
1:  procedure SV(V, E)
2:      for every vertex u ∈ V do
3:          f[u], f_next[u] ← u
4:      end for
5:      repeat
6:          ▷ Step 1: Tree hooking
7:          for every (u, v) ∈ E do in parallel
8:              if f[u] = f[f[u]] and f[v] < f[u] then
9:                  f_next[f[u]] ← f[v]
10:             end if
11:         end for
12:         f ← f_next
13:         ▷ Step 2: Shortcutting
14:         for every u ∈ V do in parallel
15:             if f[u] ≠ f[f[u]] then
16:                 f_next[u] ← f[f[u]]
17:             end if
18:         end for
19:         f ← f_next
20:     until f remains unchanged
21: end procedure
```

can handle, but also provides the most scalable connected component algorithm and the most efficient minimum spanning tree forest in the literature of distributed graph computing, showing a promising future of the linear algebra approach in graph processing.

5.1 The FastSV algorithm

In this section, we introduce four important optimizations for the simplified SV algorithm, obtaining FastSV with faster convergence.

5.1.1 A simplified SV algorithm

Algorithm 5.1 describes the simplified SV algorithm, which is the basis of our parallel algorithm. Initially, the parent $f[u]$ of a vertex u is set to itself to denote n single-vertex

trees. We additionally maintain a copy f_{next} of the parent vector so that the parallel algorithm reads from f and writes to f_{next}. Given a fixed ordering of vertices, each execution of Algorithm 5.1 generates exactly the same pointer graph after the ith iteration because of using separate vectors for reading and writing. Hence, the convergence pattern of this parallel algorithm is completely deterministic, making it suitable for massively-parallel distributed systems. By contrast, concurrent reading from and writing to a single vector f still deliver the correct connected components, but the structures of intermediate pointer graphs are not deterministic.

In each iteration, the algorithm performs tree hooking and shortcutting operations in order:

- **Tree hooking (line 6–8):** for every edge (u, v), if u's parent $f[u]$ is a root and $f[v] < f[u]$, then make $f[u]$ point to $f[v]$. As mentioned before, the updated parents are stored in a separate vector f_{next} so the updated parents are not used in the current iteration.

- **Shortcutting (line 11–13):** if a vertex u does not point to a root vertex, make u point to its grandparent $f[f[u]]$.

The algorithm terminates when the parent vector remains unchanged in the latest iteration. At termination, every tree becomes a star, and vertices in a star constitute a connected component. The correctness of this algorithm is discussed in previous work [64]. However, as mentioned before, without the unconditional hooking used in the original SV algorithm, we can no longer guarantee that Algorithm 5.1 converges in $O(\log n)$ iterations. We will show in Section 5.5 that Algorithm 5.1 indeed converges slowly, but does not require the worst case bound $O(n)$ iterations for the practical graphs we considered. Nevertheless, the extra iterations needed by Algorithm 5.1 increase the runtime of parallel SV algorithms. To alleviate this problem, we develop several novel hooking schemes, ensuring that the improved algorithm FastSV is as simple as Algorithm 5.1, but the former converges faster than the latter.

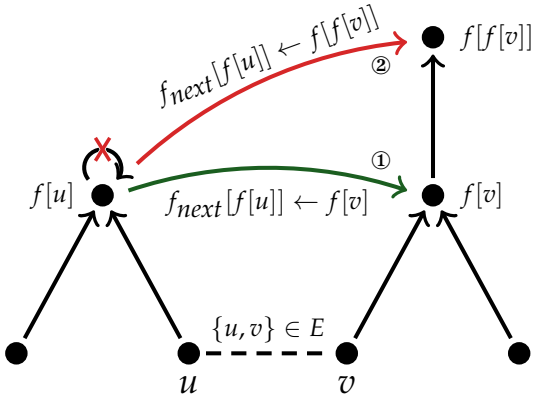


Figure 5.3: Two different ways of performing the tree hooking. (1) the original algorithm that hooks u's parent $f[u]$ to v's parent $f[v]$, (2) hook u's parent $f[u]$ to v's grandparent $f[f[v]]$. Both strategies are correct and the latter one improves the convergence.

5.1.2 Optimizations to improve convergence

Hooking to grandparent

In the original algorithm, the tree hooking is represented by the assignment $f_{next}[f[u]] \leftarrow f[v]$ (line 8 in Algorithm 5.1) requiring $f[u]$ to be a root vertex, $(u, v) \in E$ and $f[v] < f[u]$. It is not hard to see, if we perform the tree hooking using v's grandparent $f[f[v]]$, saying $f_{next}[f[u]] \leftarrow f[f[v]]$, the algorithm will still produce the correct answer. To show this, we visualize both operations in Figure 5.3.

Suppose (u, v) is an edge in the input graph and $f[v] < f[u]$. The original hooking operation is represented by the green arrow in the figure, which hooks $f[u]$ to v's parent $f[v]$. Then, our new strategy simply changes $f[v]$ to v's grandparent $f[f[v]]$, resulting the red arrow from $f[u]$ to $f[f[v]]$. It is not hard to see, as long as we choose a value like $f[f[v]]$ such that it is in the same tree of v, we can easily prove the correctness of the algorithm. One can also expect that any value like $f^k[v]$ (v's k-th level ancestor) will also work.

Intuitively, choosing a higher ancestor of v in the tree hooking will likely create shorter trees, leading to faster convergence (all trees are stars at termination). However, finding higher ancestors may incur additional computational cost. Here, we choose grandparents $f[f[v]]$ because they are needed in the shortcutting operation anyway; hence, using grandparents does not incur additional cost in the hooking operation.

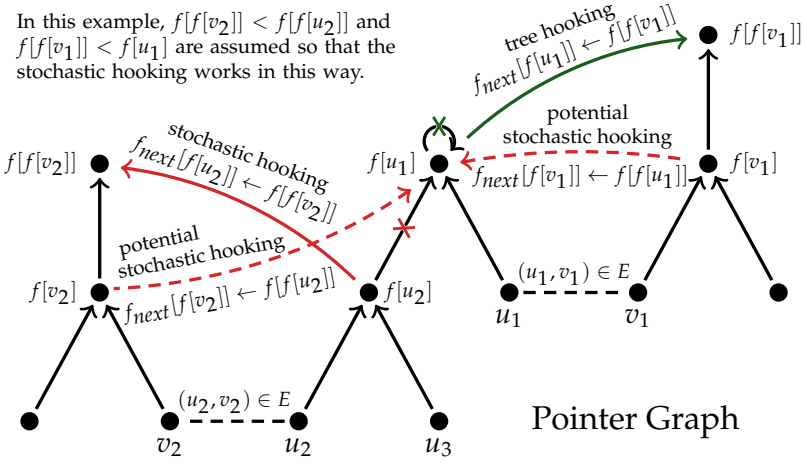


Figure 5.4: The stochastic hooking strategy. Suppose there are two edges (u_1, v_1) and (u_2, v_2) activating the hooking operation. The red arrows are the potential modifications to the pointer graph due to our stochastic hooking strategy, which tries to hook a non-root vertex to another vertex. The solid line successfully modifies $f[u_2]$'s pointer to $f[f[v_2]]$, but the dashed lines do not take effect due to the ordering on the vertices.

Stochastic hooking.

The original SV algorithm and Algorithm 5.1 always hooked the root of a tree onto another tree (see Figure 5.3 for an example). Therefore, the hooking operation in Algorithm 5.1 never breaks a tree into multiple parts and hooks different parts to different trees. This restriction is enforced by the equality check $f[f[u]] = f[u]$ in line 7 of Algorithm 5.1, which is only satisfied by roots and their children. We observed that this restriction is not necessary for the correctness of the SV algorithm. Intuitively, we can split a tree into multiple parts and hook them independently because these tree fragments will eventually be merged to a single connected component when the algorithm terminates. We call this strategy *stochastic hooking*.

The stochastic hooking strategy can be employed by simply removing the condition $f[f[u]] = f[u]$ from line 7 of Algorithm 5.1. Then, any part of a tree is allowed to hook onto another vertex when the other hooking conditions are satisfied. It should be noted that after removing the condition $f[f[u]] = f[u]$, it is possible that a tree may hook onto a vertex in the same tree. This will not affect the correctness though. In this case, the effect of stochastic hooking is similar to the shortcutting, which hooks a vertex to some other vertex with a smaller identifier.

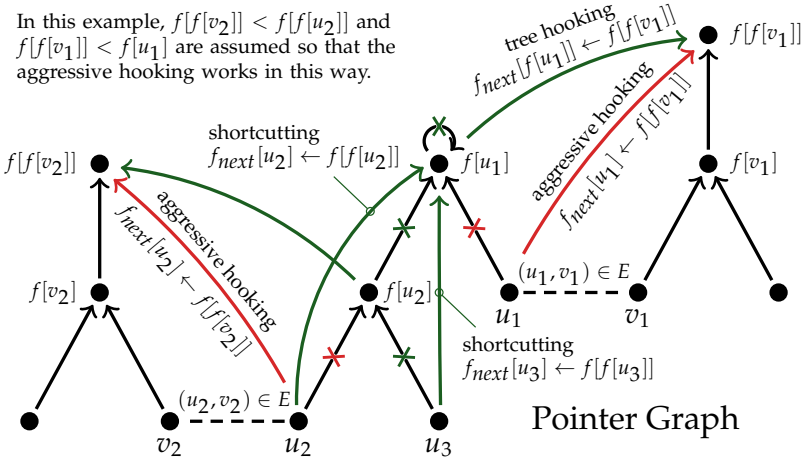


Figure 5.5: The aggressive hooking strategy. Suppose there are two edges (u_1, v_1) and (u_2, v_2) activating the hooking operation. The green arrows represent the hooking strategies introduced so far, and the red arrows represent our aggressive hooking strategy where a vertex may point at one of its neighbor's grandparent. Some vertices may have multiple arrows (like u_2), and which vertex to hook onto is decided by the ordering on the vertices.

Figure 5.4 shows an example of stochastic hooking by the solid red arrow from $f[u_2]$ to $f[f[v_2]]$. In the original algorithm, u_2 does not modify its non-root parent $f[u_2]$'s pointer, but stochastic hooking changes $f[u_2]$'s pointer to one of u's neighbor's grandparent $f[f[v_2]]$. Suppose $f[u_1]$ points to $f[f[v_1]]$ after the tree hooking, we can see that $f[u_1]$ and $f[u_2]$ might be no longer in the same connected component (assuming $f[f[v_1]]$ and $f[f[v_2]]$ are in different trees). Possible splitting of trees is a policy that differs from the conventional SV algorithm, but it gives a non-root vertex an opportunity to be hooked. In Figure 5.4, $f[u_2]$'s new parent $f[f[v_2]]$ is smaller than $f[u_1]$, which can expedite the convergence.

Algorithm 5.2 presents the high-level description of FastSV using the new hooking strategies. Here, $\overset{\min}{\longleftarrow}$ denotes a compare-and-assign operation that updates an entry of f_{next} only when the right hand side is smaller. The stochastic hooking is shown in line 6–7, and the shortcutting operation in line 12–13 is also affected by the removal of the predicate $f[f[u]] = f[u]$.

Aggressive hooking.

Next, we give a vertex u another chance to hook itself onto another tree whenever possible. This strategy is called *aggressive hooking*, performed by $f_{next}[u] \xleftarrow{\min} f[f[v]]$ for all $(u, v) \in E$. Figure 5.5 gives an example of aggressive hooking by the red arrow for u_1 and u_2. Here, u_1's pointer will not be modified by any hooking operation introduced so far. Then, the aggressive hooking makes u_1 point to its newest grandparents $f[f[v_1]]$, as if an additional shortcutting is performed. We should mention that the cost of an additional shortcutting $f'_{next} \leftarrow f_{next}[f_{next}]$ is expensive due to the recalculation of grandparents, while the aggressive hooking is essentially a cheap element-wise operation over f by reusing some results in the stochastic hooking. We will discuss how they are implemented Section 5.1.3.

For u_2 in Figure 5.5, it only performs the shortcutting operation $f_{next}[u_2] \leftarrow f[f[u_2]]$ in the original algorithm, and now the aggressive hooking performs $f_{next}[u_2] \leftarrow f[f[v_2]]$. Our implementation let $f[u_2]$ point to the smaller one between $f[f[u_2]]$ and $f[f[v_2]]$, which is expected to give the best convergence for vector f.

Early termination.

The last optimization is a generic one that applies to most variations of the SV algorithm. SV's termination is based on the stabilization of the parent vector f, which means even if f reaches the converged state (where every vertex points to the smallest vertex in its connected component), we need an additional iteration to verify that. We will see in Section 5.3.4 that for most real-world graphs, FastSV usually takes 5 to 10 iterations to converge. Hence, this additional iteration can consume a significant portion of the runtime. The removal of the last iteration is possible by detecting the stabilization of the grandparent $f[f]$ instead of f. The following lemma ensures the correctness of this new termination condition.

Lemma 5.1 *After an iteration, if the grandparent $f[f]$ remains unchanged, then the vector f will not be changed afterwards.*

The proof makes use of the following lemmas.

Lemma 5.2 *During the whole algorithm, $f[u] \leq u$ holds for all vertices u.*

Algorithm 5.2 The FastSV algorithm. **Input:** $G(V, E)$. **Output:** The parent vector f

```
1:  procedure FastSV(V, E)
2:      for every vertex u ∈ V do
3:          f[u], f_next[u] ← u
4:      end for
5:      repeat
6:          ▷ Step 1: Stochastic hooking
7:          for every (u, v) ∈ E do in parallel
8:              f_next[f[u]] ⟵(min) f[f[v]]
9:          end for
10:         ▷ Step 2: Aggressive hooking
11:         for every (u, v) ∈ E do in parallel
12:             f_next[u] ⟵(min) f[f[v]]
13:         end for
14:         ▷ Step 3: Shortcutting
15:         for every u ∈ V do in parallel
16:             f_next[u] ⟵(min) f[f[u]]
17:         end for
18:         f ← f_next
19:     until f[f] remains unchanged
20: end procedure
```

Proof. Initially, $f[u] = u$ for all vertex u and the lemma holds trivially. The the operation $\overset{\min}{\longleftarrow}$ ensures that f can only decrease, so the lemma always holds. □

Lemma 5.3 *After an iteration, if the grandparent $f[f]$ remains unchanged, then every vertex hooks onto its grandparent in the previous operation.*

Proof. By contradiction. Suppose u changes its pointer to some v other than $f[f[u]]$, then since it overrides the shortcutting operation $f_{next}[u] \overset{\min}{\longleftarrow} f[f[u]]$ we know that $v < f[f[u]]$. By Lemma 5.2, u's new grandparent $f_{next}[v] \leq v < f[f[u]]$, then the grandparent of u is changed. □

Lemma 5.4 *After an iteration, if the grandparent $f[f]$ remains unchanged, then every vertex points to a root now.*

Proof. By contradiction. Suppose u's new parent v is not a root, then u's new grandparent is $f_{next}[v] < v = f[f[u]]$ (by Lemma 5.2 and Lemma 5.3), which means u's grandparent has changed. □

Here we prove Lemma 5.1.

Proof. We show that no hooking operation will be performed if $f[f]$ remains unchanged after an iteration. The aggressive hooking in the form of $f_{next}[u] \xleftarrow{\min} f[f[v]]$ is overridden by the shortcutting operation $f_{next}[u] \xleftarrow{\min} f[f[u]]$ in the previous iteration (by Lemma 5.3), meaning that $f[f[u]] \leq f[f[v]]$ for all $(u, v) \in E$. Since then, $f[f]$ is not changed, so the aggressive hooking will not be performed in the current iteration either. The stochastic hooking $f_{next}[f[u]] \xleftarrow{\min} f[f[v]]$ will not be performed since for all $(u, v) \in E$ we have $f[f[u]] \leq f[f[v]]$. Shortcutting will not be performed either since every vertex points to a root now (by Lemma 5.4). Then, no hooking operation can be performed, and the vector f remains unchanged afterwards. □

In practice, we found that on most practical graphs, FastSV identifies all the connected components before converged, and the last iteration always performs the shortcutting operation to turn the trees into stars. In such case, the grandparent vector $f[f]$ converges one iteration earlier than f.

5.1.3 A linear algebra formulation

In GraphBLAS, we assume that the vertices are indexed from 0 to $|V|-1$, then the vertices and their associated values are stored as GraphBLAS object `GrB_Vector`. The graph's adjacency matrix is stored as a GraphBLAS object `GrB_Matrix`. For completeness, we concisely describe the GraphBLAS functions used in our implementation below, where the formal descriptions of these functions can be found in the API document [5]. We use $\emptyset$ to denote `GrB_NULL`, which is fed to those ignored input parameters.

- The function `GrB_mxv`$(y, \emptyset, \text{accum}, \text{semiring}, \mathbf{A}, x, \emptyset)$ multiplies the matrix $\mathbf{A}$ with the vector x on a semiring and outputs the result to the vector y. When the accumulator (a binary operation *accum*) is specified, the multiplication result is combined with y's original value instead of overwriting it.

- The function `GrB_extract`$(y, \emptyset, \emptyset, x, \text{index}, \text{n}, \emptyset)$ extracts a sub-vector y from the specified positions in an input vector x. We can regard this operation as $y[i] \leftarrow x[\text{index}[i]]$ for $i \in [0\,..\,n-1]$ where n is the length of the array *index* and also the vector y.

- The function `GrB_assign`$(y, \emptyset, \text{accum}, x, \text{index}, \text{n}, \emptyset)$ assigns the entries from the input vector x to the specified positions of an output vector y. We can regard it as $y[\text{index}[i]] \leftarrow x[i]$ for $i \in [0\ ..\ n-1]$ where n is the length of the array *index* and also the vector x. *accum* is the same as the one in `GrB_mxv`.
- The function `GrB_eWiseMult`$(y, \emptyset, \emptyset, \text{binop}, x_1, x_2, \emptyset))$ performs the element-wise (generalized) multiplication on the intersection of elements of two vectors x_1 and x_2 and outputs the vector y.
- The function `GrB_Vector_extractTuples`$(\text{index}, \text{value}, \&\text{n}, f)$ extracts the nonzero elements (tuples of index and value) from vector f into two separate arrays *index* and *value*. It returns the element count to n.

For the rest functions, we have `GrB_Vector_dup` to duplicate a vector, `GrB_reduce` to reduce a vector to a scalar value through a user-specified binary operation, and `GrB_Matrix_nrows` to obtain the dimension of a matrix.

Algorithm 5.3 describes the FastSV algorithm in GraphBLAS. Before every iteration, we calculate the initial grandparent gf for every vertex. First, we perform the stochastic hooking in line 10–11. GraphBLAS has no primitive that directly implements the parallel-for on an edge list (line 9 in Algorithm 5.2), so we have to first aggregate v's grandparent $gf[v]$ to vertex u for every $(u, v) \in E$, obtaining the vector $mngf[u] = \min_{v \in N(u)} gf[v]$. This can be implemented by a matrix-vector multiplication $mngf = \mathbf{A} \cdot gf$ using the (select2nd, min) semiring. Next, the hooking operation is implemented by the assignment $f[f[u]] \leftarrow mngf[u]$ for every vertex u. This is exactly the `GrB_assign` function in line 10 where the indices are the values of vector f extracted in either line 5 before the first iteration or line 16 from the previous iteration. The accumulator `GrB_MIN` prevents the nondeterminism caused by the modification to the same entry of f, and the minimum operation gives the best convergence in practice.

Aggressive hooking is then implemented by an element-wise multiplication $f \leftarrow \min(f, mngf)$ in line 13. Although it is another operation in FastSV that performs the parallel-for on an edge list, it can reuse the vector $mngf$ computed in the previous step, so the aggressive hooking is actually efficient. Shortcutting is also implemented by an the element-wise multiplication $f \leftarrow \min(f, gf)$ in line 15. Next, we calculate the grandparent vector $gf[u] \leftarrow f[f[u]]$. It is implemented by the `GrB_extract` function in line 18 where the indices are the values of f extracted in line 17.

Algorithm 5.3 The linear algebra FastSV algorithm. **Input:** The adjacency matrix **A** and the parent vector f. **Output:** The parent vector f.

```
1:  procedure FastSV(A, f)
2:      GrB_Matrix_nrows (&n, A)
3:      GrB_Vector_dup (&gf, f) ▷ initial grandparent
4:      GrB_Vector_dup (&dup, gf) ▷ duplication of gf
5:      GrB_Vector_dup (&mngf, gf)
6:      GrB_Vector_extractTuples (index, value, &n, f)
7:      Sel2ndMin ← a (select2nd, Min) semiring
8:      repeat
9:          ▷ Step 1: Stochastic hooking
10:         GrB_mxv (mngf, ∅, GrB_MIN, Sel2ndMin, A, gf, ∅)
11:         GrB_assign (f, ∅, GrB_MIN, mngf, value, n, ∅)
12:         ▷ Step 2: Aggressive hooking
13:         GrB_eWiseMult (f, ∅, ∅, GrB_MIN, f, mngf, ∅)
14:         ▷ Step 3: Shortcutting
15:         GrB_eWiseMult (f, ∅, ∅, GrB_MIN, f, gf, ∅)
16:         ▷ Step 4: Calculate grandparents
17:         GrB_Vector_extractTuples (index, value, &n, f)
18:         GrB_extract (gf, ∅, ∅, f, value, n, ∅)
19:         ▷ Step 5: Check termination
20:         GrB_eWiseMult (diff, ∅, ∅, GxB_ISNE, dup, gf, ∅)
21:         GrB_reduce (&sum, ∅, Add, diff, ∅)
22:         GrB_assign (dup, ∅, ∅, gp, GrB_ALL, 0, ∅))
23:     until sum = 0
24: end procedure
```

At the end of each iteration, we calculate the number of modified entries in gf in line 20 – 21 to check whether the algorithm has converged or not. A copy of gf is stored in the vector *dup* for determining the termination in the next iteration.

5.1.4 A distributed implementation using CombBLAS

The distributed version of FastSV is implemented in CombBLAS [4]. CombBLAS provides all operations needed for FastSV, but its API differs from the GraphBLAS standard. GraphBLAS's *collections* (matrices and vectors) are opaque datatypes whose internal representations (sparse or dense) are not exposed to users, but CombBLAS distinguishes them in the user interface. Then, GraphBLAS's functions often consist of multiple operations (like masking, accumulation and the main operation) as described

in Section 5.1.3, while in CombBLAS we usually perform a single operation at a time. Despite these differences, a straightforward implementation of FastSV on CombBLAS can be obtained by transforming each GraphBLAS function to the semantically equivalent ones in CombBLAS, using dense vectors in all scenarios.

The parallel complexity of the main linear algebraic operations used in FastSV (the vector variants of `GrB_extract` and `GrB_assign`, and the `GrB_mxv`), as well as the potential optimizations are discussed in the LACC paper [31]. Due to the similarity of FastSV and LACC in the algorithm logic, they can be optimized by the similar optimization techniques. We briefly summarize them below.

Broadcasting-based implementation for the extract and assign operations. The *extract* and *assign* operations fetch or write data on the specified locations of a vector, which may cause a load balancing issue when there is too much access on a few locations. In FastSV, these locations are exactly the set of parent vertices in the pointer graph, and due to the skewed structure of the pointer graph, the root vertices (especially those belonging to a large component) will have extremely high workload. When using the default *assign* and *extract* implementations in CombBLAS via all-to-all communication, several processes become the bottleneck and slow down the whole operation significantly. The solution is a manual implementation of these two operations via the detection of the hot spots and broadcasting the entries on those processes.

Taking advantage of the sparsity. The matrix-vector multiplication $mngf = \mathbf{A} \cdot gf$ is an expensive operation in FastSV (see our performance profiling in Figure 5.2.6). The straightforward implementation naturally chooses the sparse-matrix dense-vector (SpMV) multiplication, since all the vectors in FastSV are dense vectors. Alternatively, we can use an incremental implementation by computing $\Delta mngf = \mathbf{A} \cdot (\Delta gf)$, where $\Delta gf = gf - gf_{prev}$ containing only the modified entries of gf is stored as a sparse vector, so the multiplication is the sparse-matrix sparse-vector multiplication (SpMSpV) [65]. Depending on the sparsity of Δgf, SpMSpV could have much lower computation and communication cost than SpMV. We use a threshold on the portion of modified entries of gf to decide which method to use in each iteration, which effectively reduces the computation time. Figure 5.2.6 presents a detailed evaluation.

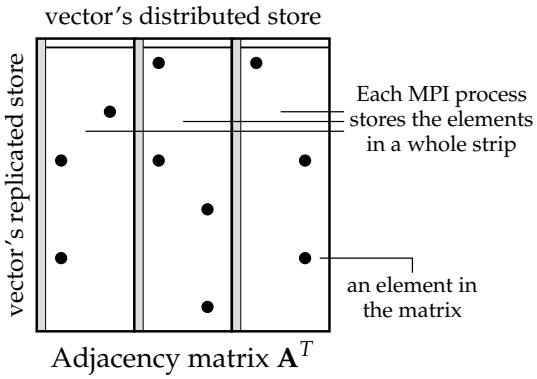


Figure 5.6: Data placement of the matrix and vector. The matrix is divided by column so that each MPI process stores the elements on a whole strip in the figure. For the vectors, they can be either replicated on all the processes, or split into disjoint segments and distributed on all the processes.

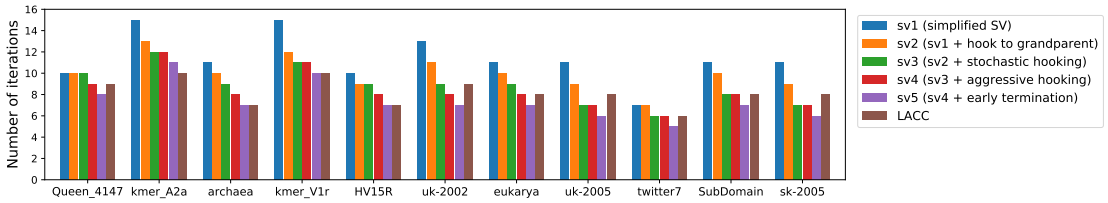


Figure 5.7: Number of iterations the simplified SV takes after performing each of the optimizations (sv5 is exactly our FastSV algorithm), and the number of iterations LACC takes.

5.2 Evaluation of FastSV

In this section, we evaluate various aspects of FastSV showing its fast convergence, shared- and distributed-memory performance, scalability and several other performance characteristics. We compare FastSV with LACC [31] that has demonstrated superior performance over other distributed-memory parallel CC algorithms. Table 5.1 shows a diverse collection of large graphs used to evaluate CC algorithms. To the best of out knowledge, the Hyperlink graph [67] with 3.27B vertices and 124.90B edges is the largest publicly available graph.

5.2.1 Evaluation platform

We evaluate the performance of distributed algorithms on NERSC Cori supercomputer. Each node of Cori has Intel KNL processor with 68 cores and 96GB of memory.

Table 5.1: Graph datasets used to evaluate the parallel connected component algorithms.

Graph	Vertices	Edges	CCs	Description
Queen_4147	4.15M	166.82M	1	3D structural problem [50]
kmer_A2a	170.73M	180.29M	5353	Protein k-mer graphs from GenBank [50]
archaea	1.64M	204.78M	59794	archaea protein-similarity network [66]
kmer_V1r	214.01M	232.71M	9	Protein k-mer graphs, from GenBank [50]
HV15R	2.02M	283.07M	1	Computational Fluid Dynamics Problem [50]
uk-2002	18.48M	298.11M	1990	2002 web crawl of .uk domain [50]
eukarya	3.24M	359.74M	164156	eukarya protein-similarity network [66]
uk-2005	39.45M	936.36M	7727	2005 web crawl of .uk domain [50]
twitter7	41.65M	1.47B	1	twitter follower network [50]
SubDomain	82.92M	1.94B	246969	1st-level subdomain graph extracted from Hyperlink [67]
sk-2005	50.64M	1.95B	45	2005 web crawl of .sk domain [50]
MOLIERE_2016	30.22M	3.34B	4457	automatic biomedical hypothesis generation system [50]
Metaclust50	282.20M	37.28B	15982994	similarities of proteins in Metaclust50 [66]
Hyperlink	3.27B	124.90B	29360027	hyperlink graph extract from the Common Crawl [67]

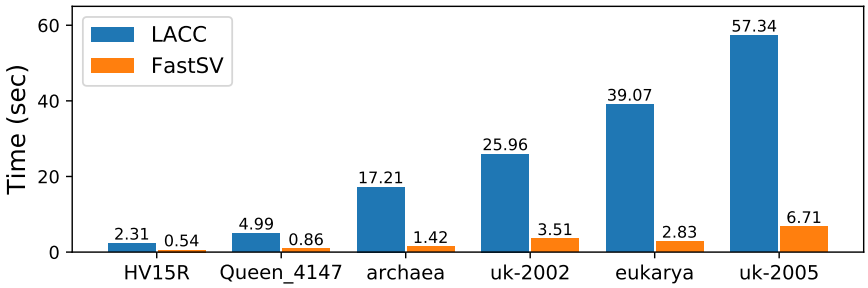


Figure 5.8: Performance of the parallel FastSV and LACC in SuiteSparse:GraphBLAS on six small graphs.

All operations in CombBLAS are parallelized with OpenMP and MPI. Given p MPI processes, we always used a square $\sqrt{p} \times \sqrt{p}$ process grid. In our experiments, we used 16 threads per MPI process. The execution pattern of our distributed algorithm follows the bulk synchronous parallel (BSP) model, where all MPI processes perform local computation followed by synchronized communication rounds.

We also show the shared-memory performance of FastSV in the SuiteSparse:GraphBLAS library [27]. These experiments are conducted on Amazon EC2's r5.4xlarge instance (128G memory, 16 threads).

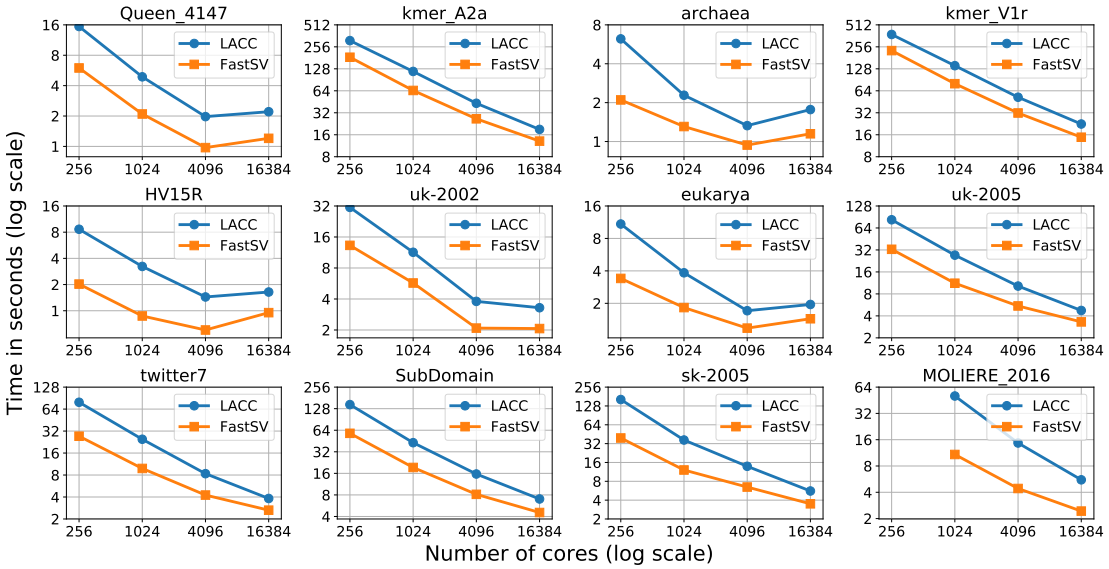


Figure 5.9: Strong scaling of distributed-memory FastSV and LACC using up to 16384 cores (256 nodes).

5.2.2 Speed of convergence

At first, we show how different hooking strategies impact the convergence of SV and FastSV algorithms. We start with the simplified SV algorithm (Algorithm 5.1) and incrementally add different hooking strategies as shown in Figure 5.7. The rightmost bars report the number of iterations needed by LACC.

Figure 5.7 shows that the simplified SV without unconditional hooking can take up to 1.57× more iterations than LACC. We note that despite needing more iterations, Algorithm 5.1 can run faster than LACC in practice because each iteration of the former is faster than each iteration of the latter. Figure 5.7 demonstrates that SV converges faster as we incrementally apply advanced hooking strategies. In fact, every hooking strategy improves the convergence of some graphs, and their combination improves the convergence of all graphs. Finally, the early termination discussed in Section 5.1.2 always removes an additional iteration needed by other algorithms. With all improvements, sv5 which represents Algorithm 5.2, on average reduces 35.0% iterations (min 20%, max 46.2%) from Algorithm 5.1. Therefore, FastSV converges as quickly as, or faster than, LACC.

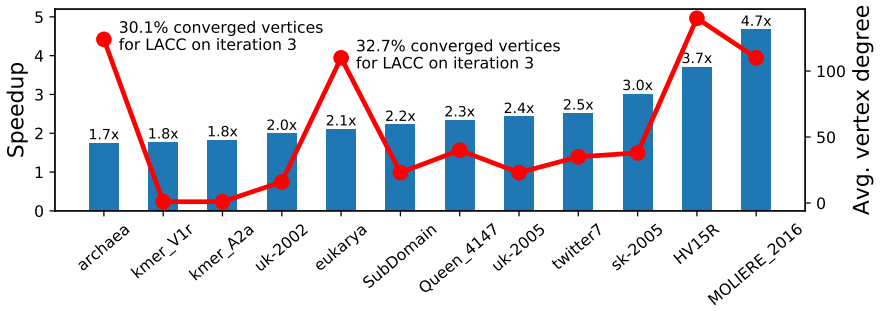


Figure 5.10: The speedup of FastSV over LACC on twelve small datasets using 256 cores (bar chart) and each graph's density in terms of average vertex degree (line chart). A positive correlation between the two metrics can be observed, except for the two outliers *archaea* and *eukarya*.

5.2.3 Shared-memory performance

To check the correctness of Algorithm 5.3, we implemented it in SuiteSparse:GraphBLAS, a multi-threaded implementation of the GraphBLAS standard. LACC also has an unoptimized SuiteSparse:GraphBLAS implementation available as part of the LAGraph library [68]. We compare the performance of Figure 5.8 and LACC in this setting on an Amazon EC2's r5.4xlarge instance with 16 threads. Figure 5.8 shows that FastSV is significantly faster than LACC (avg. 8.66×, max 13.81×). Although both algorithms are designed for distributed-memory platforms, we still observe better performance of FastSV, thanks to its simplicity.

5.2.4 Distributed-memory performance

We now evaluate the performance of FastSV implemented using CombBLAS and compare its performance with LACC on the Cori supercomputer. Both algorithms are implemented in CombBLAS, so they share quite a lot of common operations and optimization techniques (see Section 5.1.4), making it a fair comparison between the two algorithms. Generally, FastSV operates with simpler computation logic and uses less expensive parallel operations than LACC. However, depending on the structure of the graph, LACC can detect the already converged connected components on the fly and can potentially use more sparse operations. Hence, the structure of the input graph often influences the relative performance of these algorithms.

Figure 5.9 summarizes the performance of FastSV and LACC on twelve small

datasets. We observe that both FastSV and LACC scale to 4096 cores on all the graphs, and for the majority of the graphs (8 out of 12), they continue scaling to 16384 cores. The four graphs on which they stop scaling are just too small that both algorithms finish within 2 seconds. FastSV outperforms LACC on all instances. On 256 cores, FastSV is 2.80× faster than LACC on average (min 1.66×, max 4.27×). When increasing the number of nodes, the performance gap between FastSV and LACC shrinks slightly, but FastSV is still 2.53×, 1.97× and 1.61× faster than LACC on average on 1024, 4096 and 16384 cores, respectively.

To see how the performance of FastSV and LACC are affected by the graph structure, we plot the average degree ($|E|/|V|$) and the speedup of FastSV over LACC for each graph (using 1024 cores) in Figure 5.10. Generally, FastSV tends to outperform LACC by a significant margin on denser graphs. This is mainly due to fewer matrix-vector multiplications used in FastSV, whose parallel complexity is highly related to the density of the graph. The outliers *archaea* and *eukarya* are graphs with a large number of small connected components: they have more than 30% converged vertices detected early. On such graphs, LACC's detection of converged connected components provides it with better opportunities to employ sparse operations, while such detection is not allowed in FastSV. Nevertheless, LACC's sparsity optimization still cannot compensate its high computational cost in each iteration.

5.2.5 Strong scalability

We separately analyze the performance of FastSV and LACC on the two largest graphs in Table 5.1. Hyperlink is perhaps the largest publicly available graph, making it the largest connectivity problem we can currently solve. Since each of these two graphs requires more than 1TB memory, it may be impossible to process them on a typical shared-memory server. Figure 5.11 shows the strong scaling of both algorithms and the better performance of FastSV. On the smaller graph Metaclust50, both algorithms scale to 65,536 cores where FastSV is 1.47× faster than LACC. On the Hyperlink graph containing 124.9 billion edges, they continue scaling to 262,144 cores, where FastSV achieves an 2.03× speedup over the LACC algorithm.

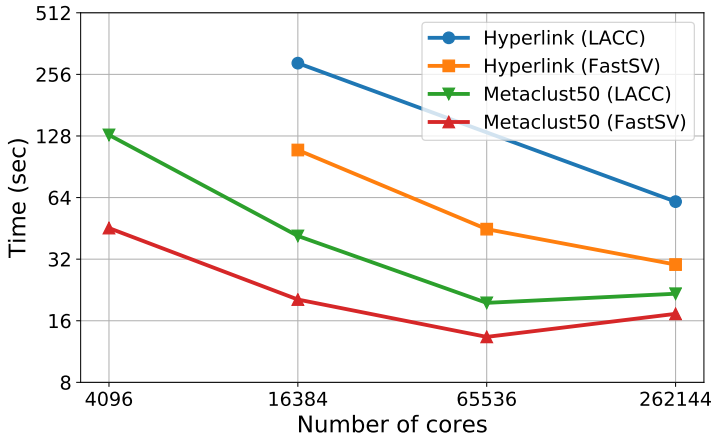


Figure 5.11: Performance of FastSV and LACC with two large graphs on CoriKNL (up to 262144 cores using 4096 nodes).

5.2.6 Performance characteristics

In this subsection, we present more aspects of FastSV to help readers understand the characteristics of this algorithm.

Performance breakdown.

Figure 5.12 shows the execution time of FastSV by breaking the runtime into three parts: finding the grandparent, matrix-vector multiplication, the hooking operations. The time spent on checking the termination is omitted, since it is insignificant relative to other operations. Each of these operations contributes significantly to the total execution time. Finding the grandparent and the hooking operations basically reflect the parallel complexity of the `extract` and `assign` operations, and the ratio of them is relatively stable for all graphs. By contrast, the execution time of SpMV varies considerably across different graphs, because SpMV's complexity depends on the density of a graph.

Execution time reduced by the sparsity optimization.

As mentioned in Section 5.1.4, FastSV dynamically selects SpMV or SpMSpV based on the changes in the grandparent vector gf. This optimization is particularly effective for high-density graphs where SpMV usually dominates the runtime (see Figure 5.12). Figure 5.13 explains the benefit of sparsity with four representative graphs, where we plot the number of vertices modified in each iteration. We observe that only a small fraction of vertices participates in the last few iterations where SpMSpV can be used

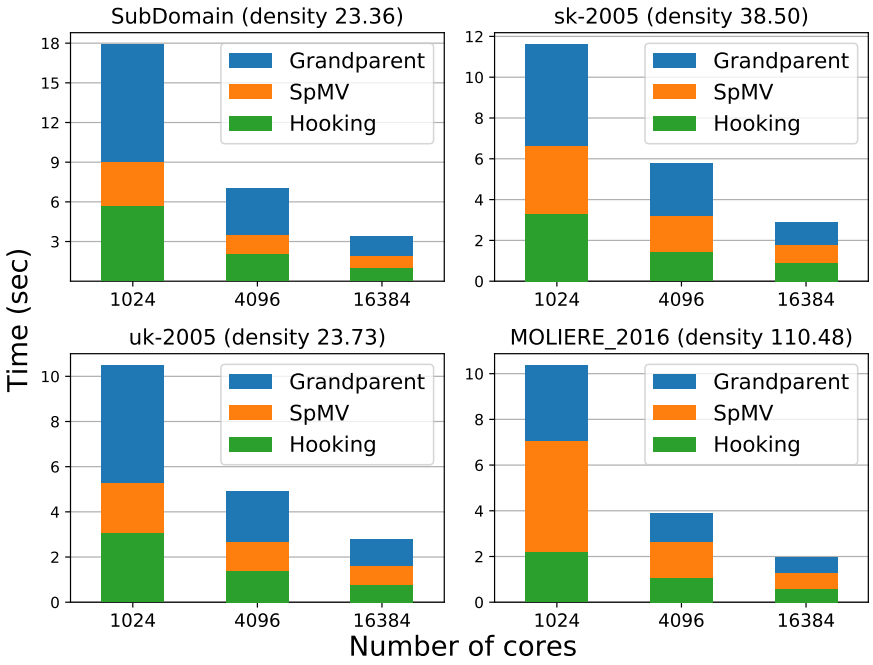


Figure 5.12: Performance breakdown of FastSV on four representative graphs.

instead of SpMV. As shown by the red runtime lines in Figure 5.13, the use of SpMSpV drastically reduces the runtime of the last few iterations.

5.3 Boruvka's algorithm

5.3.1 Vertex-centric Boruvka's Algorithm

Boruvka's algorithm [18] is considered the easiest for parallelization among the classic MSF algorithms [18, 69, 70]. Conventionally, its parallelization on distributed-memory is based on the Pregel [1] model where the computation is encoded as a *vertex-program* that runs on every vertex and exchanges messages with the other vertex in a bulk synchronous way [9]. Boruvka's algorithm proceeds in rounds, and in each round, it picks a set of edges in a greedy manner and adds them to the final results. The edges selected so far constitute a forest, in the edge of each round we contract the graph by removing the edges inside each tree. The algorithm terminates when all the edges are removed. It is worth noting that to quickly check whether an edge is connecting two vertices in the same tree or not, we encode all the spanning trees in a forest as

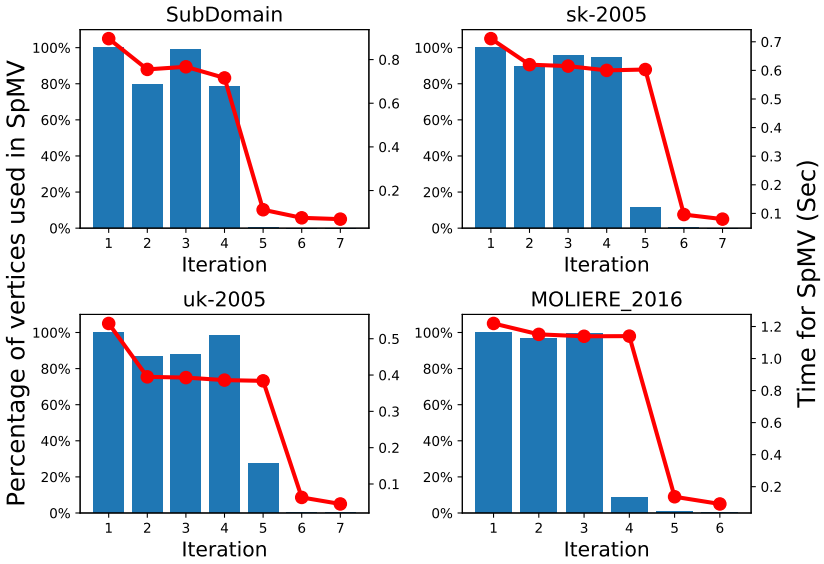


Figure 5.13: Percentage of vertices that participate in the SpMV (sparse matrix-vector multiplication) operation for each iteration (bar chart), and the runtime for SpMV (line chart). A vertex participates the SpMV if its grandparent gf is not changed in the previous iteration.

a vector f, as shown in Figure 5.14. Each spanning tree elects a root vertex (a.k.a. *supervertex* [71]), and the vector f stores the parent vector for every vertex (the root points to itself). By flattening this structure, we have f stores each vertex's supervertex so that we can easily tell whether two vertices are in the same spanning tree or not.

Boruvka's algorithm starts with n supervertices each containing a singe vertex. In each round, we perform the following steps to construct the minimum spanning tree.

- **Min-edge picking:** each supervertex selects the minimum edge from the outgoing edges of all the vertices in the spanning tree. This selected edge is then a candidate of the final result[1]. This procedure is further divided into two steps:
  - **Step 1:** every vertex u finds the minimum ordered pair $(w, f[v])$ for all v in u's adjacency list. Then, u sends the ordered pair along with the selected edge (u, v) to u's supervertex $f[u]$.
  - **Step 2:** every supervertex then picks the minimum ordered pair $(w, f[v])$ among the received edges and changes its pointer to the supervertex $f[v]$.

[1]The same edge might be picked by the supervertices of both endpoints. In this situation we only keep one of them and add it to the final result.

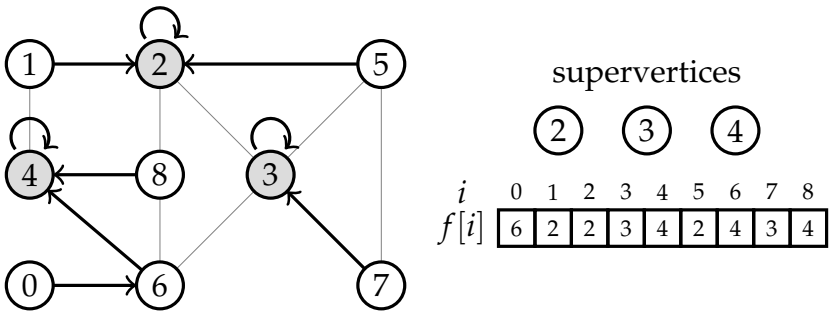


Figure 5.14: Encode the currently recognized connected components of a graph as a vector. Each connected component has a unique identifier called *supervertex*, and all the vertices connect to the supervertex through a direct link or a path.

If the spanning tree is isolated, the supervertex does nothing.

- **Supervertex selection:** after the min-edge picking, the non-isolated supervertices in the graph will constitute a set of conjoined-trees [71], each being a directed tree with a cycle of length two on the top. We identify every such cycle (u, v), select the smaller one of u and v as the new supervertex and make it a new root by creating a self-loop.

- **Flattening:** every vertex u finds the supervertex by repeatedly performing the shortcutting operation $f[u] \leftarrow f[f[u]]$ until the vector f stabilizes.

- **Edge cleaning:** remove the edges inside each tree, which are $(u, v) \in E$ having $f[u] = f[v]$. Repeat the above steps if there are still edges remained.

Figure 5.15 shows the MSF computation on a weighted graph using Boruvka's algorithm. Starting from the input graph, every supervertex (which has no child in the first iteration) selects the minimum edge and the arrows form three conjoined-trees. Then, the vertices 2, 3, 4 are selected as the new supervertices. The flattening operation makes every vertex directly point to the new supervertex, and the edge cleaning removes the edges inside each spanning tree. The graph is neither shrunk nor relabeled in order to maintain a low computation and communication cost. From the second iteration, the min-edge picking needs every supervertex to select the minimum outgoing edge from the whole spanning tree, and the result is shown in the lower left figure. The other operations are similar to the first iteration, generating a spanning tree containing all the vertices in the graph. The algorithm then terminates.

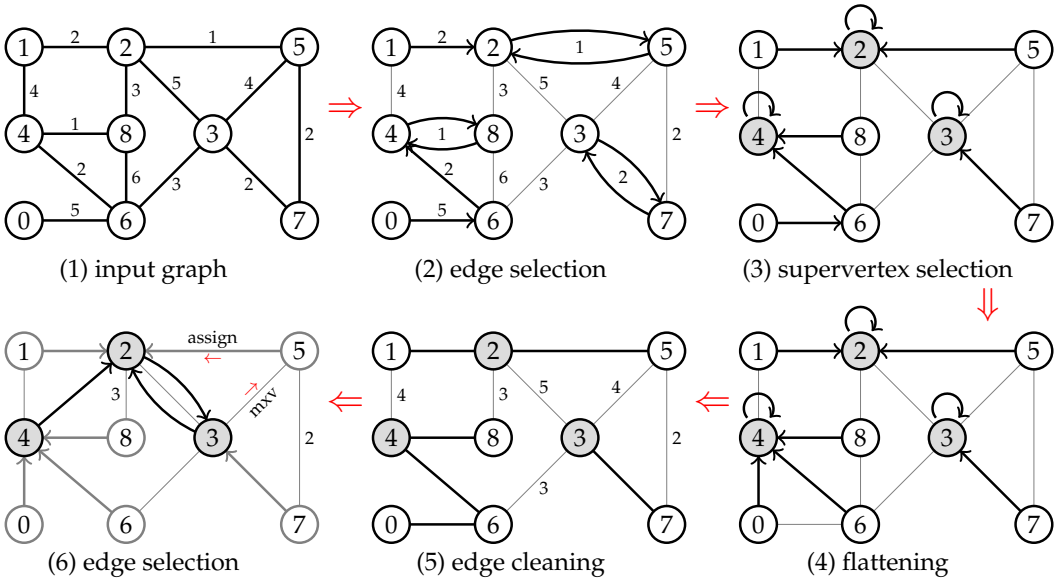


Figure 5.15: The MSF computation using Boruvka's algorithm, from the beginning to the min-edge picking in the second iteration. The arrows represent the parent vector f.

The primary issue affecting the distributed-memory performance in Pregel-like systems is the edge cleaning, which requires the predicate $f[u] = f[v]$ to be evaluated on every edge (u, v). In these systems, vector f is an attribute associated on every vertex, and the evaluation of $f[u] = f[v]$ cannot be done without the message exchange if u and v are assigned to different physical nodes. By assuming a random vertex assignment, the expected amount of messages to complete the edge cleaning is $O(E)$ [72]. Our evaluation results show that it is a significant overhead in the MSF computation that takes up to 81.86% of the total runtime (see Section 5.5.3).

5.3.2 A linear algebra formulation

In this section, we express Boruvka's algorithm in the language of linear algebra. A commonly used linear-algebraic API for graph computation is the GraphBLAS C standard [5], but it is slightly verbose and hides the insights for distributed computing. We therefore define a set of simplified APIs and then describe Boruvka's algorithm in our framework.

In Algorithm 5.4, we present a step-by-step implementation for Borvuka's algorithm using the linear algebra APIs presented before. The input is an undirected weighted

graph represented as an adjacency matrix **A** and the output **MSF** is a submatrix of **A** containing the selected edges in the final minimum spanning forest.

First of all, the min-edge picking is implemented on line 10–15, which aims to let every supervertex know (a) the minimum outgoing edge in the whole tree, including the edge weight and the supervertex id on the other side, and (b) the original edge in matrix **A** in order to recover the minimum spanning forest. Most of the job is accomplished by the multiplication of **A** and x shown in Figure 5.1, and by combining the result y with every u's own vertex id, we obtain the vector e_v containing the edge selected by every vertex. Next, every supervertex collects the minimum edge from the whole spanning tree using the `assign` operation, and the idea of this operation is illustrated in Figure 5.2 using a simplified example. Finally, if $e_s[r] \neq \infty$ for a supervertex r, it modifies its entry in vector f on line 15 using the second field in e_s.

The supervertex selection on line 16–18 identifies the pairs of vertices u and v that point to each other ($f[u] = v$ and $f[v] = u$) and selects the smaller one as the new supervertex. By a simple substitution, we need to find every u having $f[f[u]] = u$ and let u point to itself if $u < f[u]$. The grandparent vector $f[f]$ is computed by the `extract` function and is stored in the vector g, then the following two `eWiseMult` operations compute all possible u's and changes the pointer if the condition $u < f[u]$ is satisfied.

The flattening operation on line 21–25 is first performed on the old supervertices through the repetition of the shortcutting operation $f \leftarrow f[f]$ until stabilized. This procedure takes at most $O(\log h)$ rounds where h is the maximum height of all the spanning trees. On line 25, all the vertices (which point to an old supervertex already) perform a single round of shortcutting to see the new supervertex.

The last step is the edge cleaning on line 27–33. We first pick the edges in **A** that constitute the minimum spanning tree, by simply inspecting the vertices that become a non-supervertex in this iteration. Then, on line 28–31, we count the new supervertices (excluding the isolated supervertices in the previous step) and terminate the algorithm instantly if there is at most one active supervertices remained. Next, by the `select` operation, we remove the edges inside each spanning tree and obtain a smaller matrix **A**. The algorithm terminates when **A** is empty.

5.3.3 A simplified algorithm for computing CCs

Boruvka's algorithm is capable of finding all the connected components in the graph, since each spanning tree is exactly a connected component, and the parent vector f stores the component identifier for each vertex when terminated. In order to make the CC computation more efficient, we can remove all the unnecessary computations in Algorithm 5.4. The tree construction on line 27 is obviously useless. Then, in the edge picking, we no longer take the edge weight into consideration, so the supervertices only collects the smallest neighboring supervertex id from the whole tree. We give a reference implementation in Algorithm 5.5.

5.3.4 The guarantee of convergence

Boruvka's algorithm terminates in $\log(n)$ rounds in the worst case for both MSF and CC computation. Consider the number of non-isolated spanning trees in the graph. Initially, this number is n, and in each round after the min-edge picking, all the non-isolated supervertices constitute a set of conjoined-trees, and only one vertex in each conjoined-tree will become a new supervertex in the next round. A conjoined-tree has at least two vertices, so the number of new supervertices (including both isolated and non-isolated ones) is at most half of the number of non-isolated supervertices in the previous round. In practice, this algorithm converges very fast. Section 5.5 presents more detailed results for different types of graphs. A very useful corollary is that the accumulated number of non-isolated supervertices during the whole computation is $O(n)$.

5.4 Implementation

5.4.1 Data placement for matrix and vector

The graph is represented as an n-by-n symmetric matrix $\mathbf{A}$ representing the adjacency relationship of the vertices. We split the matrix by column and let each process store a whole vertical strip, as shown in Figure 5.6. We assume that the vertices are indexed with $0..n-1$ where n is the total number of vertices. Then, the elements stored on each process are the edges having the destination vertex in a disjoint range of $[0..n-1]$. We

randomly permute the indices to obtain a good load balancing.

For each vector, it is either fully distributed on all the processes such that each process holds a disjoint segment with roughly the same number of elements, or simply replicated on all the processes. The replication of a vector obviously has higher memory consumption, but we consider it viable since the number of vertices in a practical graph can typically fit into the main-memory, and for some operations (like the flattening operation we will discuss later), only in this representation it can be finished in reasonable time. Choosing the most suitable representation for each vector used in Boruvka's algorithm is important to achieve high performance.

5.4.2 Overview of the implementation

In this thesis, we implement all our linear algebraic operations using two levels of data parallelism, one is MPI for inter-node communication and the other is OpenMP for multithreading on each node. We mainly discuss the following operations, `mxv`, `assign` and `select`, since the others are essentially element-wise operations that can be easily parallelized. We also give a special implementation of the flattening operation (which iteratively performs the `extract` operation), which makes use of the vertex replications to avoid the high communication cost.

In our implementation, we replicate as few vectors as possible to minimize the memory consumption. Essentially, only the vector f and e_s in Figure 5.6 need to be replicated on all the processes, and it is basically an requirement for implementing the `mxv` and `assgin` operations. For the constant vectors (such as *inf*, *all* and *ind*) and the intermediate vectors x and g, even though they are involved in the element-wise computations with f, in our implementation, we compute them on the fly so that the actual storage is not necessary.

5.4.3 Parallelization of the linear algebra operations

In this subsections, we go through the parallel implementation for all the aforementioned linear algebra operations.

- `mxv`: this is the key linear algebra operation in many graph computations. In our model, the multiplication $y = x\mathbf{A}^T$ (or equivalently $y = \mathbf{A}x$ since we do not

distinguish a vector with its transpose) is accomplished by replicating the vector x on every process and computing a distributed vector y. Each element of y is obtained by a process computing the inner product of x and the corresponding column in $\mathbf{A}^T$ using the user-specified operators $\oplus$ and $\otimes$. We store the matrix $\mathbf{A}^T$ in the compressed sparse column (CSC) format so that the OpenMP parallelism is straightforward.

- `assign`: this is the key operation to let every supervertex collect the information from all the vertices in the spanning tree. Our implementation is presented in Figure 5.16 where the output vector y is the only vector that is initially replicated on all the processes. To accomplish the `assign` operation, each process first performs the local computation that modifies its own replica of y using OpenMP and atomic writes, then all the processes synchronize the modified entries in y. For the synchronization, the sparse vector `super`, playing the role of the mask of `assign`, is replicated on all the processes through the `MPI_Allgatherv` function, and then the specified entries in y are extracted and synchronized through the `MPI_Allreduce` function.

- `flattening`: the flattening operation (line 21–24 of Algorithm 5.4) is an iterative procedure that performs the `extract` operation repeatedly to half the distance from every vertex in *super* to the root. Here the boolean vector *super* contains the old supervertices generated in the previous iteration. The overall computation cost reaches $O(np)$ (see the analysis in Section 5.4.4) but the communication cost is completely removed, making it an efficient solution on small clusters.

- `select`: in Boruvka's algorithm, this operation removes the edges $(u, v) \in E$ having $f[u] = f[v]$. Due to the replication of the vector f on all the processes, the `select` operation can inspect the supervertex of both u and v without incurring any communication cost. We implement this operation by a single-pass scan on every edge, and we allow a vertex u to keep different neighbors belonging to the same connected component, so that an $O(E)$ computation cost can be ensured. The `mxv` operation is implemented with `select` to maximize the performance.

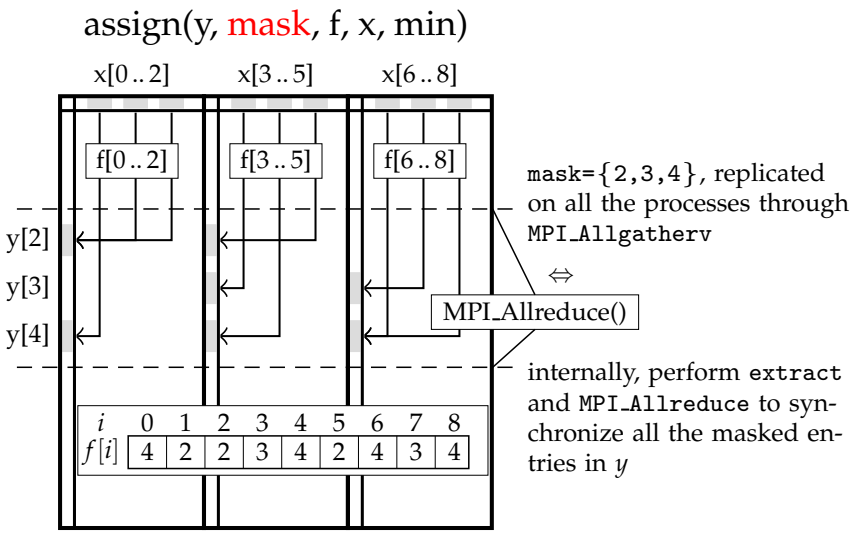


Figure 5.16: The implementation of the *assign* operation in our model. Each process sees a disjoint portion of the input vectors x, f and *mask*. Then, it computes and stores the partial results to the replicated vector y. We first replicate the vector *mask* using MPI's all-gather function, then all the processes synchronize the union of the modified entries in y (indicated by the parameter *mask*) by MPI's all-reduce function.

5.4.4 Cost Analysis

In the end of this section, we analyze the total computation and communication cost for our implementation of Boruvka's algorithm. We use n and m to represent the number of vertices and edges in the graph, and use p to represent the number of processes. The algorithm terminates in $O(\log n)$ rounds.

First, the only communication cost in our implementation is the `assign` operation where the vertices sends their selected edges to the supervertex. The total communication cost, as illustrated in Figure 5.16, is the cost of performing an `MPI_Allgather` to replicate the mask on all the processes plus an `MPI_Allreduce` to synchronize the values of the supervertices. We know that the accumulated number of supervertices during the whole computation is $O(n)$ (see Section 5.3.4), so the `assign` operation requires $O(np)$ messages in total for both operations.

For all the other operations, there is no communication cost at all. In each iteration, `mxv` and `select` take $O(m)$ time, and all the element-wise operations, depending on whether the vector is replicated or not, takes either $O(np)$ or $O(n)$ time where p is the replication factor. Note that an $O(np)$ time complexity can be very huge if we

Table 5.2: Graph datasets used to evaluate the parallel CC and MSF algorithms.

Graph	Vertices	Edges	Description
GAP-road	23.95M	57.71M	the distances of all of the roads in the USA [73]
GAP-twitter	61.58M	2.937B	a crawl of Twitter's social network in 2009 [74]
GAP-web	50.64M	3.861B	a web crawl of .sk domain in 2005 [50]
GAP-kron	134.2M	4.223B	a graph synthesized by the Kronecker synthetic graph generator [75]
GAP-urand	134.2M	4.295B	a graph synthesized by the Erdos–Reyni model (Uniform Random) [76]

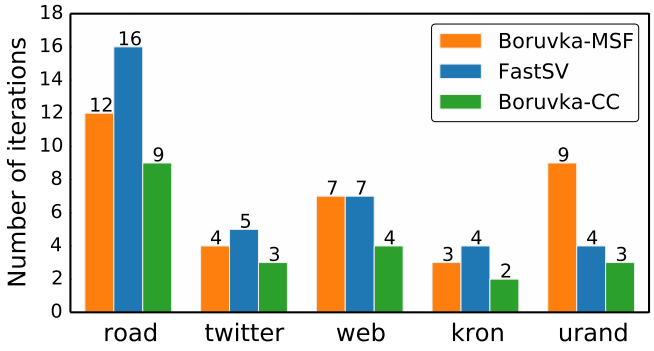


Figure 5.17: Number of iterations Boruvka's algorithm and FastSV take on the graphs in the GAP benchmark.

unrestrainedly increase the number of processes p for the CC or MSF computation. In order to achieve high performance, we use as few processes as possible in practice to reduce the computation cost. We assume $M > n$ where M is the memory space of a single node, and using at most $p = O(m/n)$ physical nodes the graph can fit into the main-memory of the cluster.

For the flattening operation, in each Boruvka round, every process individually performs the subgraph flattening (line 21–24) which scans the supervertices only by at most $O(\log n)$ times. The overall time complexity is $O(pn \log n)$ where n is the accumulated number of supervertices during the whole computation, p is the replication factor. By summing up all the numbers above, Boruvka's algorithm has an $O((m + pn) \log n)$ computation cost and $O(np)$ communication cost. In particular, when $p = 1$, there is actually no communication cost, and our parallel implementation for Boruvka's algorithm achieves the optimal $O(m \log n)$ time complexity.

5.5 Evaluation of Boruvka's algorithm

In this section, we evaluate various aspects of our linear algebraic Boruvka's algorithm, including the speed of convergence, distributed-memory performance, scalability and several other performance characteristics. To make a fair comparison with previous works, we target the minimum spanning forest problem and the connected component problem separately, and our solution for solving each problem is named as Boruvka-CC and Boruvka-MSF, respectively. Boruvka-CC is basically a simplification of Boruvka-MSF that only reports the number of connected components, ignoring the edge weight and the construction of the minimum spanning tree.

For the minimum spanning forest problem, Boruvka's algorithm is currently the only algorithm parallelized on the distributed-memory architecture, and previous works [8, 28] all adopt the vertex-centric paradigm. We compare our linear algebraic Boruvka-MSF with Pregel-channel [28]'s implementation, which reports the fastest runtime compared to previous works. There are more distributed-memory algorithms [77, 78, 31, 17] proposed for distributed-memory architecture in various models, and we compared our Boruvka-CC with FastSV [17], which is currently the fastest and the most scalable algorithm using the linear algebra abstraction.

We use the GAP benchmark matrices [79] (summarized in Table 5.2) to evaluate these implementations, which include both synthetic and real-world weighted graphs of different types.

5.5.1 Evaluation platform

We evaluate the distributed-memory algorithms on Amazon EC2. Our implementation is compared against two libraries, CombBLAS [4] and Pregel-channel [28]. CombBLAS use both MPI and OpenMP to parallelize the graph algorithms, but it adopts a different partitioning strategy that the matrix is decomposed in a two-dimensional way and the vectors are fully distributed on all the processes using a range-based manner. Pregel-channel is parallelized with MPI only, in which the vertices are distributed to all the processes in a hash-based manner and the edges are associated on the source vertex. All these implementations follow the Bulk Synchronous Parallel (BSP) [9] model where all MPI processes perform local computation followed by synchronized communication

round. We set up a cluster of eight nodes with 10Gbps network bandwidth, and each node is an r5.4xlarge instance having 16 vCPUs and 128G memory. This is the minimum configuration for the CombBLAS library in terms of memory space. We compile all the programs using g++ 7.4.0 with -O3 flag.

5.5.2 Speed of convergence

We first look at the speed of convergence of the Boruvka-CC and the FastSV algorithm. FastSV's design is based on the PRAM Shiloach-Vishkin algorithm [16] with various optimizations to improve its convergence on distributed-memory architecture. Then, Figure 5.17 clearly shows that Boruvka-CC takes less number of iterations than FastSV on all our five graphs. The number of iterations does not directly reflect the performance since the complexity of each iteration matters, but later we will see the high efficiency of Boruvka's algorithm's each iteration as well. Then, computing the minimum spanning tree using Boruvka's algorithm generally takes more iterations, and the main reason is the different edge picking strategy of Boruvka-MSF that takes the edge weight into consideration. There is also a vertex-centric implementation of Boruvka's algorithm in the vertex-centric paradigm, and we should not that it takes exactly the same number of iterations as Boruvka-MSF, since the algorithm is deterministic and the implementation methods will not affect its convergence.

5.5.3 Distributed-memory performance

Next, we evaluate the distributed-memory performance of our algorithms in solving the connected component and the minimum spanning forest problems. The experiment is conducted on a cluster of eight nodes using 64 cores in total.

Minimum spanning forest. We first compare the runtime of Boruvka's MSF algorithm's vertex-centric implementation with our linear algebraic implementation. We use Pregel-channel's implementation [28] of Boruvka's algorithm due to its channel mechanism to reduce the message size and the higher performance reported in the paper compared to the previous works. Pregel-channel's main functionality runs in the pure MPI mode, so we launch 64 MPI processes to make full use of the CPU cores. For our Boruvka-MSF implementation in linear algebra, we launch 8 MPI processes, one

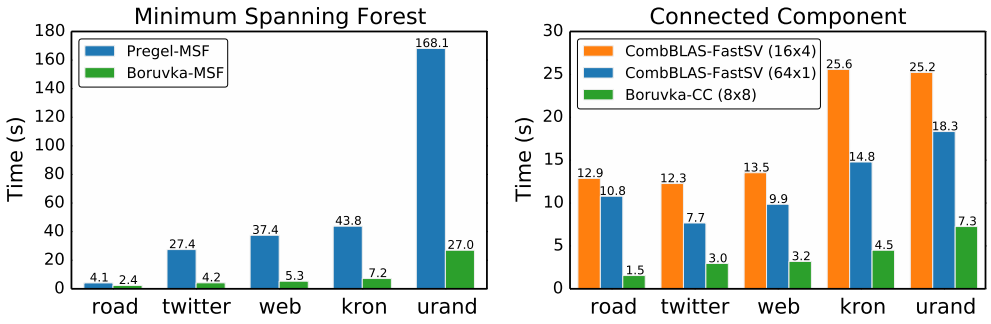


Figure 5.18: Runtime of Boruvka's algorithm on the five graphs in the GAP benchmark.

for each node, each using 8 threads for its local computation. Figure 5.18 presents the results on the five graphs in the GAP benchmark.

Compared to Pregel-channel's vertex-centric implementation, our linear algebraic implementation achieves a speedup of 1.75× on the road graph and a speedup of 6.09× to 7.03× on the other four graphs. Our implementation of Boruvka's algorithm has a factor of p (the number of processes) for element-wise operations on replicated vectors. Therefore, on the planner road graph with high sparsity, our approach is not remarkably better than a vertex-centric implementation. However, for the other graphs where $|E|$ is one or two magnitude larger than $|V|$, we can see that the linear algebraic approach is significantly faster, and the reason is our efficient implementation of the `select` operation that incurs no communication cost at all. We note that the percentage of time spent on the edge cleaning operation in Pregel reaches 75.13% to 81.86% on the last four graphs (except the road graph), proving that the edge cleaning is a very heavy operation in the vertex-centric model. On average[2], our linear algebra implementation achieves a 6.10× speedup on all the graphs in the GAP benchmark.

Connected component. We then compare the runtime of FastSV and Boruvka-CC for finding the connected components in each graph. FastSV is implemented in CombBLAS [4] using its linear algebra primitives while the CombBLAS is parallelized by MPI and OpenMP. CombBLAS's implementation requires the number of MPI processes p to be a square number so that the matrix is divided into $\sqrt{p}\times\sqrt{p}$ submatrices and is evenly distributed to the process grid. We therefore run CombBLAS's FastSV program in two configurations, one with 64 MPI processes each having 1 thread, and

[2]The averaged speedup is the ratio of the total time spent on all the graphs in the GAP benchmark by the two programs.

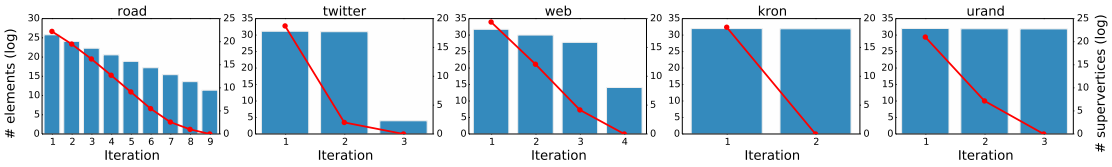


Figure 5.19: Number of elements in the matrix at the beginning of each iteration (bar chart) and the number of newly selected supervertices in each iteration (line chart) for Boruvka-CC.

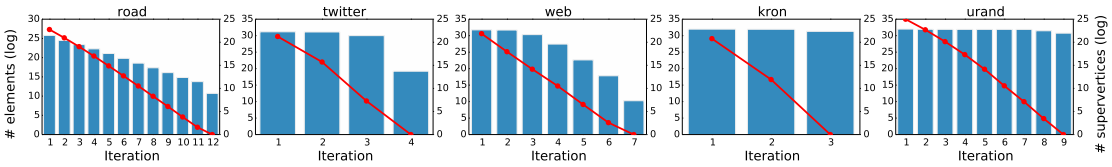


Figure 5.20: Number of elements in the matrix at the beginning of each iteration (bar chart) and the number of newly selected supervertices in each iteration (line chart) for Boruvka-MSF.

the other with 16 MPI processes each having 4 threads. Figure 5.18 presents the results on the five graphs in the GAP benchmark.

The linear algebra Boruvka-CC is clearly faster than FastSV's both runs on all instances. For FastSV, running it in the pure MPI mode is faster than the hybrid MPI+OpenMP mode in CombBLAS. Compared to FastSV's best result using 64 processes, our Boruvka-CC achieves a speedup of 6.98× on the road graph, and it is 2.53× to 3.29× faster on the other graphs. We have already seen that Boruvka-CC converges faster than FastSV, but even if we compare the runtime of each iteration, Boruvka-CC is still 3.20× faster on the road graph and is 1.56× to 1.89× on the other graphs, due to our slightly better `mxv` and much better implementation of the `extract` and `assign` operation. We present a detailed analysis in Section 5.5.4. On average, Boruvka-CC is 3.16× faster than FastSV on all the graphs in the GAP benchmark using 8 nodes and in total 64 threads.

5.5.4 Performance characteristics

In this subsection, we present more aspects of Boruvka's algorithm to help readers understand the strategies used in our distributed-memory implementation.

Active edges and supervertices. In Figure 5.19 and Figure 5.20, we plot the

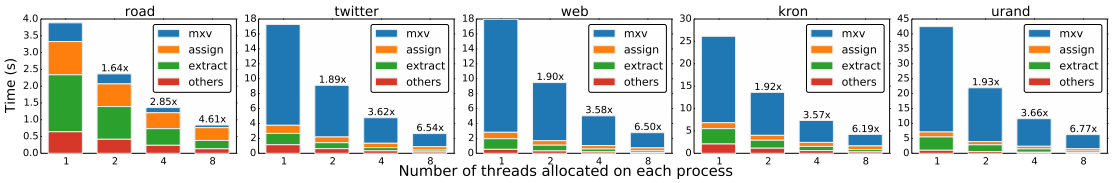


Figure 5.21: Breakdown of the runtime for Boruvka-CC on eight nodes using different number of threads.

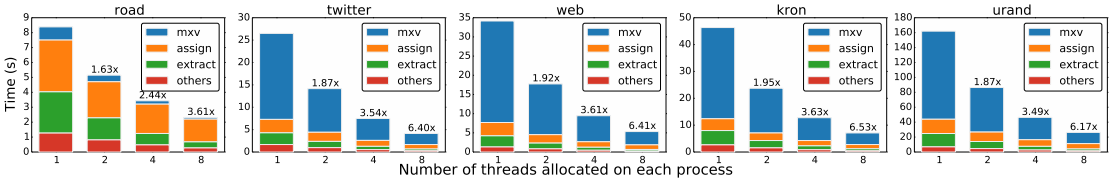


Figure 5.22: Breakdown of the runtime for Boruvka-MSF on eight nodes using different number of threads.

number of active edges and supervertices during the execution of Boruvka-CC and Boruvka-MSF. The active edges are the edges remained in the graph in the end of each iteration, meaning that they are currently connecting different spanning trees in the spanning forest constructed so far. Then, the active supervertices are the newly selected supervertices in each iteration, which decides the communication cost of the `assign` operation. We use bar chart and line chart respectively to represent the active edges and supervertices and both numbers are presented in log scale.

The number of active supervertices, as expected, decreases rapidly with the execution of Boruvka-MSF and Boruvka-CC. The accumulated number during the whole computation (except the first iteration that all vertices are supervertices), as shown in Table 5.3, varies from 0.28% to 23.92% for Boruvka-CC and 1.35% to 41.25% for Boruvka-MSF on different types of graphs. It reflects the actual communication cost of the `assign` operation.

On the contrary, the number of active edges in each iteration may not notably decrease during the computation. On the two synthetic graphs kron and urand, even in the last iteration, the number of edges removed is less than 4%. Therefore, in the worst case, Boruvka's algorithm has an average $O(E)$ edges in the matrix during the whole computation. A vertex-centric implementation therefore needs $O(E)$ messages to perform the `select` operation in every iteration, while in our linear algebraic implementation there is no communication at all. We have already seen that the edge

Table 5.3: Accumulated number of active supervertices during the whole computation of the Boruvka's algorithm for CC and MSF, presented in the ratio to the graph size n.

algorithm	road	twitter	web	kron	urand
Boruvka-CC	23.92%	0.71%	1.38%	0.28%	1.56%
Boruvka-MSF	41.25%	4.14%	8.09%	1.35%	31.05%

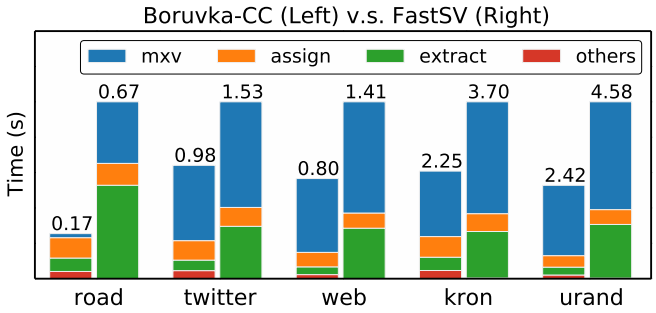


Figure 5.23: Per-iteration runtime for each type of linear algebra operations in Boruvka-CC and FastSV.

cleaning can take up to 81.86% percentage of the total execution time in a Pregel implementation.

Scalability and performance breakdown. To see the scalability of Boruvka's algorithm, we change the number of threads each process can use in the CC and MSF computation. The total number of processes is still 8 since we have 8 nodes in the cluster. We break down the runtime of Boruvka's algorithm into four parts, the matrix-vector multiplication along with the `select` operation (they are combined into a single operation), the `assign`, the `extract` and all the other operations. Figure 5.21 and Figure 5.22 presents our results.

The overall speedup with the OpenMP parallelism using 8 threads achieves more than 6× for both Boruvka-CC and Boruvka-MSF on the four dense graphs. The `mxv` dominates the runtime and achieves the highest speedup (more than 7.5× on average), while `assign` has the lowest speedup (around 2-3×) due to the constant communication cost. The `extract` and the others are element-wise operations and scale as well as `mxv`.

Figure 5.23 shows the per-iteration time spent on each operation for Boruvka-CC and FastSV. On average, Boruvka-CC's `mxv` is 1.66× faster and `assgin` is equally fast, but the `extract` function (or more precisely the iterative `extract` in flattening) is 5.51× faster than FastSV. The main reason is the removal of the communication cost of the `extract` operations in our implementation. For the `mxv` operation, Boruvka's

algorithm pays additional cost for the `select` operation, but on graphs like road and web, the cost of `mxv` decreases dramatically in later iterations.

5.6 Related work

Finding the minimum spanning tree (or forest) in an undirected weight graph is a well-studied problem in sequential or on shared-memory. The classic sequential algorithms given by Boruvka [18], Prim [70] and Kruskal [69] provide $O(m \log n)$ time complexity. Some algorithms [80, 81, 82, 83, 84, 85] have theoretically lower time complexity, but they are either hard to parallelize or not efficient on practical graphs due to a big hidden factor. On multiprocessors or shared-memory systems, Bader's work [86] presents a new MST algorithm that combines the idea of Prim's and Boruvka's algorithm, and several recent works on parallelizing Boruvka's algorithm include [87, 88, 89] exploits various issues on shared-memory, including the graph representation, cache effect and the load balancing. Our work focuses on reducing the communication cost on distributed-memory, but on shared-memory, it also has the optimal time complexity $O(m \log n)$ and we demonstrate its high scalability in our evaluation.

On distributed-memory, Boruvka's algorithm has been parallelized and implemented in several Pregel-like systems [90, 8, 28]. Pregel [1] stands for a particular type of vertex-centric system based on the Bulk-synchronous parallel (BSP) model [9] and inter-vertex messages passing. We note that the vertex-centric graph analytics systems are not limited to the Pregel-like systems (see the surveys [91, 92]). However, the majority of them (e.g., [10, 11, 12, 13, 14, 15]) are functionally similar to the sparse-matrix vector multiplication, while Pregel's model is yet the only model that can deal with both edge removal and non-neighborhood communication (analogous to the `select`, `assign` and `extract` operations in linear algebra), which are necessary for Boruvka's algorithm and some other scalable graph computations [17, 31, 32]. We should note that none of the existing linear algebra graph libraries [4, 33, 13] have fully implemented these operations, and our work is the first linear algebra parallelization of Boruvka's algorithm on distributed-memory.

Boruvka's algorithm is intrinsically vertex-centric, and Salihoglu's work [71] mainly verified two effective strategies called storing the edges at sub-vertices (SEAS)

and edge cleaning on-demand (ECOD). Then, to optimize the communication-intensive edge cleaning operation, Yan's work [21] introduces an elegant and effective approach called the *request-respond* paradigm, and later Zhang's work [28] proposes the channel mechanism to reduce the overall communication cost by allowing heterogeneous messages to be used and separately optimized in the same program. However, the key problem of high communication cost does not disappear, and it is inevitable in Pregel's model due to its hash-based vertex partition. Our work is the first attempt to use linear algebra to parallelize Boruvka's algorithm and we provide message-efficient implementations for several key operations based on our novel vertex replication strategy.

For the connected component (CC) problem, on shared-memory there are both theoretically efficient algorithm using low-diameter decomposition (LLD) [93] and empirically efficient solutions like direction-optimized BFS [94, 95] and union-find using concurrent disjoint-set data structure [96]. Compared to these works, Boruvka's algorithm does not provide the optimal solution. However, they are not efficient on distributed-memory due to the high latency of accessing data on remote nodes. A feasible solution on distributed-memory is the multi-source parallel BFS [97] (a.k.a. label propagation), but it converges slowly on large-diameter graphs. Then, there are scalable CC algorithms [8, 77, 31, 17] based on the PRAM Shiloach-Vishkin (SV) [16] and Awerbuch-Shiloach (AS) algorithm [98] with performance guarantees (empirical or theoretical $O(\log n)$ rounds). We are the first to use Boruvka's algorithm for finding connect components on distributed-memory. We show that Boruvka's algorithm converges even faster than the state-of-the-art FastSV [17] algorithm and achieves a 3.16× speedup using 64 cores.

Algorithm 5.4 The linear algebra Boruvka's algorithm for finding minimum spanning tree. **Input:** The adjacency matrix **A**. **Output:** The matrix **MSF** containing the edges in the solution.

```
1:  procedure BoruvkaMSF(A)
2:      fill(super, true) ▷ non-isolated supervertex
3:      fill(inf, ∞); fill(all, true) ▷ constant vectors
4:      fillWithIndex(ind) ▷ vertex id (also a constant vector)
5:      fillWithIndex(f) ▷ the initial parent vector
6:      MSF ← ∅ ▷ return value
7:      repeat
8:          copy(f', all, f)
9:          ▷ Step 1: min-edge picking
10:         eWiseMult(x, all, f, ind, zip) ▷ x[u] = (f[u], u)
11:         fill(y, ∞); mxv(y, A, x, zip, min)
12:         eWiseMult(e_v, all, y, ind, zip) ▷ e_v[u] = (w, f[v], v, u)
13:         fill(e_s, ∞); assign(e_s, super, f, e_v, min)
14:         eWiseMult(super, all, e_s, inf, not_equal_to)
15:         apply(f, super, e_s, second)
16:         ▷ Step 2: supervertex selection
17:         extract(g, super, f, f, min)
18:         eWiseMult(mask, all, g, ind, equal_to)
19:         eWiseMult(f, mask, f, ind, min)
20:         ▷ Step 3: flattening
21:         extract(g, super, f, f)
22:         while f ≠ g do
23:             copy(f, super, g)
24:             extract(g, super, f, f)
25:         end while
26:         extract(f, all, f, f)
27:         ▷ Step 4: edge cleaning
28:         MSF ← MSF ∪ RecoverMSF(A, f', f, e_s)
29:         eWiseMult(mask, all, f, ind, equal_to)
30:         eWiseMult(super, all, super, mask, land)
31:         reduce(active, all, super, plus, 0)
32:         if (active ≤ 1) then break
33:         select(A, A, λ(i, j, a_ij) → {f[i] = f[j]})
34:         getNVals(nvals, A)
35:     until nvals = 0
36:     return MSF
37: end procedure
```

Algorithm 5.5 The simplified edge picking operation in Boruvka's algorithm for finding connected components.

```
1: procedure EdgePicking-CC(...)
2:     ▷ e_v, e_s are vectors of supervertex id
3:     fill(e_v, ∞); mxv(e_v, A, f, second, min)
4:     fill(e_s, ∞); assign(e_s, super, f, e_v, min)
5:     eWiseMult(super, all, e_s, inf, not_equal_to)
6:     copy(f, super, e_s)
7: end procedure
```

6 Conclusions and Future Work

With the increasing demand of analyzing large-scale graphs generated by the modern applications, lots of research has been invested in distributed graph systems in order to processes massive graphs efficiently in memory. This thesis gives a comprehensive overview of the existing graph analytics systems on distributed-memory in terms of three goals of such systems: (a) the distributed-memory performance and scalability and (b) the ease of the programming interface, and our goal is to have a graph analytics system that fulfill both goals.

The mainstream graph analytics systems can be categorized into two classes, the vertex-centric paradigm and the linear algebra approach, each having their own drawbacks when evaluated using the two criteria. The vertex-centric paradigm is yet the most popular and general approach for distributed graph processing, but Pregel's low-level programming interface including the message passing and state transition makes it unfriendly to users. Even with the domain-specific languages (DSLs) to ease Pregel programming, the high complexity in Pregel's optimization techniques still make it difficult for ordinary users to quickly develop efficient graph analytics

applications. The linear algebra approach, on the other hand, has a concise high-level programming interface using standardized matrix and vector operations, but the lack of a graph abstraction as well as the use of atomic linear algebra APIs make it difficult to further optimize a graph computation.

In this thesis, our main contribution is a graph analytics framework following the vertex-centric paradigm that is both user-friendly and highly efficient. Our framework is built on three key techniques, a more expressive high-level domain-specific language (DSL) called Palgol to ease vertex-centric programming by hiding the message passing from users, an efficient back end that can easily combine various optimizations in the same vertex-program, and a novel cost-based compilation technique to compile our DSL to the back end. The resultant framework has a friendly programming interface and achieve comparable performance to the state-of-the-art vertex-centric system .

We are also interested in improving graph analytics in the language of linear algebra. We currently focus on the linear algebra formulation of distributed graph algorithms and their efficient implementation, and this thesis presents a novel connected component (CC) algorithm called FastSV, and an efficient linear algebraic implementation of Boruvka's minimum spanning forest (MSF) on distributed-memory. We show that the linear algebra approach is capable of dealing with complex graph algorithms like FastSV and Boruvka's algorithm, and both of our works significantly outperform the state-of-the-art CC and MSF implementation on distributed-memory.

In the future, we are definitely interested in fully exploiting the capability of the linear algebra approach, including implementing more interesting graph algorithms in GraphBLAS and improving their efficiency on distributed-memory. An important question remained in linear algebra is whether we can build a GraphBLAS compliant and highly efficient graph analytics system for distributed-memory. We believe that a GraphBLAS compliant graph system on distributed-memory is very beneficial, since it allows the same program to be executed on different platforms, which can greatly reduce the development cost of large-scale graph applications for ordinary users. Currently, there are several linear algebra libraries [38, 4, 13] proposed for distributed-memory but none of them is GraphBLAS compliant since the standard is relatively new. Furthermore, there are still technical issues in achieving this, since an efficient graph computation on distributed-memory usually requires various optimization techniques to reduce the communication cost, but when written in

GraphBLAS API using atomic matrix or vector operations, those optimizations are hard to derive due to the lack of graph semantics. We have summarized this issue in detail in Section 1.2.2.

A possible direction to achieve our goal – ensuring high efficiency for distributed graph computations written in GrpahBLAS API – is to recover the graph semantics though program analysis. A successful framework is our SQL-core language that captures the vertex-centric graph using relational queries, and in the future we are interested in finding the connection between the relational model and the linear algebra API, so that we can eventually derive an efficient implementation of linear algebra graph computation on top of the Pregel-channel system.

Bibliography

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

[2] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):25, 2015.

[3] Da Yan, Yingyi Bu, Yuanyuan Tian, Amol Deshpande, et al. Big graph analytics platforms. *Foundations and Trends® in Databases*, 7(1-2):1–195, 2017.

[4] Aydın Buluç and John R Gilbert. The combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[5] Tim Mattson, David Bader, Jon Berry, Aydin Buluc, Jack Dongarra, Christos Faloutsos, John Feo, John Gilbert, Joseph Gonzalez, Bruce Hendrickson, et al. Standards for graph algorithm primitives. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–2. IEEE, 2013.

[6] Carl Yang, Aydin Buluc, and John D Owens. Graphblast: A high-performance linear algebra-based graph framework on the gpu. *arXiv preprint arXiv:1908.01407*, 2019.

[7] Aydin Buluc, Timothy Mattson, Scott McMillan, Jose Moreira, and Carl Yang. The GraphBLAS C API specification. *GraphBLAS. org, Tech. Rep.*, 2017.

[8] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment*, 7(14):1821–1832, 2014.

[9] Leslie G Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[10] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *arXiv preprint arXiv:1204.6078*, 2012.

[11] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[12] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, page 1. ACM, 2015.

[13] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11):1214–1225, 2015.

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

[15] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 752–768, 2018.

[16] Y. Shiloach and Uzi Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[17] Yongzhe Zhang, Ariful Azad, and Zhenjiang Hu. Fastsv: A distributed-memory connected component algorithm with fast convergence. In *SIAM Conference on Parallel Processing for Scientific Computing (PP20)*, pages 46–57. SIAM, 2020.

[18] Sun Chung and Anne Condon. Parallel implementation of Borůvka's minimum spanning tree algorithm. In *IPPS*, pages 302–308. IEEE, 1996.

[19] Sungpack Hong, Semih Salihoglu, Jennifer Widom, and Kunle Olukotun. Simplifying scalable graph processing with a domain-specific language. In *CGO*, page 208. ACM, 2014.

[20] Kento Emoto, Kiminori Matsuzaki, Akimasa Morihata, and Zhenjiang Hu. Think like a vertex, behave like a function! A functional DSL for vertex-centric big graph processing. In *ICFP*, pages 200–213. ACM, 2016.

[21] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1307–1317, 2015.

[22] Semih Salihoglu and Jennifer Widom. GPS: a graph processing system. In *SSDBM*, number 22. ACM, 2013.

[23] Nguyen Thien Bao and Toyotaro Suzumura. Towards highly scalable pregel-based graph processing platform with x10. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 501–508. ACM, 2013.

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

[25] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. GoFFish: A sub-graph centric framework for large-scale graph analytics. *arXiv preprint arXiv:1311.5949*, 2013.

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

[27] SuiteSparse:GraphBLAS – Graph algorithms in the language of linear algebra. http://faculty.cse.tamu.edu/davis/GraphBLAS.html.

[28] Yongzhe Zhang and Zhenjiang Hu. Composing optimization techniques for vertex-centric graph processing via communication channels. In *33rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 428–438. IEEE, 2019.

[29] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[30] Semih Salihoglu and Jennifer Widom. Optimizing graph algorithms on Pregel-like systems. *PVLDB*, 7(7):577–588, 2014.

[31] Ariful Azad and Aydın Buluç. LACC: A linear-algebraic algorithm for finding connected components in distributed memory. In *Proceedings of the IPDPS*, pages 2–12. IEEE, 2019.

[32] Semih Salihoglu and Jennifer Widom. Help: High-level primitives for large-scale graph processing. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6, 2014.

[33] Michael J Anderson, Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Theodore L Willke, and Pradeep Dubey. GraphPad: Optimized graph primitives for parallel and distributed platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 313–322. IEEE, 2016.

[34] Fan Yang, Jinfeng Li, and James Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *Proceedings of the VLDB Endowment*, 9(5):420–431, 2016.

[35] Harold N Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. System Sci.*, 30(2):209–221, 1985.

[36] Yongzhe Zhang, Hsiang-Shang Ko, and Zhenjiang Hu. Palgol: A high-level DSL for vertex-centric graph processing with remote data access. In *Proceedings of the 15th Asian Symposium on Programming Languages and Systems*, pages 301–320. Springer, 2017.

[37] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Scalable simd-efficient graph processing on gpus. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT '15, pages 39–50, 2015.

[38] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proceedings of the Ninth IEEE International Conference on Data Mining*, pages 229–238. IEEE, 2009.

[39] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[40] Alessandro Lulli, Patrizio Dazzi, Laura Ricci, and Emanuele Carlini. A multi-layer framework for graph processing via overlay composition. In *European Conference on Parallel Processing*, pages 515–527. Springer, 2015.

[41] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *PVLDB*, 5(8):716–727, 2012.

[42] Jie Yan, Guangming Tan, Zeyao Mo, and Ninghui Sun. Graphine: programming graph-parallel computation of large natural graphs for multicore clusters. *IEEE Transactions on Parallel and Distributed Systems*, 27(6):1647–1659, 2016.

[43] Graphx. http://spark.apache.org/graphx/.

[44] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[45] Semih Salihoglu and Jennifer Widom. HelP: High-level primitives for large-scale graph processing. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6. ACM, 2014.

[46] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3):549–587, 1990.

[47] David J DeWitt and Robert Gerber. *Multiprocessor hash-based join algorithms*. University of Wisconsin-Madison, Computer Sciences Department, 1985.

[48] Florian Bourse, Marc Lelarge, and Milan Vojnovic. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1456–1465. ACM, 2014.

[49] Jérôme Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013.

[50] Timothy A Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[51] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th International Conference on World Wide Web*, pages 107–117. Elsevier, 1998.

[52] Apache Giraph. http://giraph.apache.org/.

[53] Michael Lesniak. Palovca: describing and executing graph algorithms in Haskell. In *PADL*, pages 153–167. Springer, 2012.

[54] Onofre Coll Ruiz, Kiminori Matsuzaki, and Shigeyuki Sato. s6raph: vertex-centric graph processing framework with functional interface. In *FHPC*, pages 58–64. ACM, 2016.

[55] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In *ASPLOS*, pages 349–362. ACM, 2012.

[56] Graph 500 – Large-scale benchmarks. https://graph500.org/.

[57] Ying Xu, Victor Olman, and Dong Xu. Clustering gene expression data using a graph-theoretic approach: an application of minimum spanning trees. *Bioinformatics*, 18(4):536–545, 2002.

[58] Stijn Marinus Van Dongen. *Graph clustering by flow simulation*. PhD thesis, 2000.

[59] K Kayser, H Stute, and M Tacke. Minimum spanning tree, integrated optical density and lymph node metastasis in bronchial carcinoma. *Analytical cellular pathology: the journal of the European Society for Analytical Cellular Pathology*, 5(4):225–234, 1993.

[60] Mariel Brinkhuis, Gerrit A Meijer, LT Schuurmans, JP Baak, et al. Minimum spanning tree analysis in advanced ovarian carcinoma. an investigation of sampling methods, reproducibility and correlation with histologic grade. *Analytical and quantitative cytology and histology*, 19(3):194–201, 1997.

[61] Xue Dong Yang. An improved algorithm for labeling connected components in a binary image. Technical report, Cornell University, 1989.

[62] Xiaochun Wang, Xiali Wang, and D Mitchell Wilkes. A divide-and-conquer approach for minimum spanning tree-based clustering. *IEEE Transactions on Knowledge and Data Engineering*, 21(7):945–958, 2009.

[63] Caiming Zhong, Duoqian Miao, and Ruizhi Wang. A graph-theoretical clustering method based on two rounds of minimum spanning trees. *Pattern Recognition*, 43(3):752–766, 2010.

[64] John Greiner. A comparison of parallel algorithms for connected components. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, pages 16–25. ACM, 1994.

[65] Ariful Azad and Aydin Buluç. A work-efficient parallel sparse matrix-sparse vector multiplication algorithm. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 688–697. IEEE, 2017.

[66] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. HipMCL: a high-performance parallel implementation of the markov clustering algorithm for large-scale networks. *Nucleic Acids Research*, 46(6):e33–e33, 2018.

[67] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. Graph structure in the web—revisited: a trick of the heavy tail. In *Proceedings of the 23rd international conference on World Wide Web*, pages 427–432. ACM, 2014.

[68] Tim Mattson, Timothy A Davis, Manoj Kumar, Aydin Buluc, Scott McMillan, José Moreira, and Carl Yang. Lagraph: A community effort to collect graph algorithms built on top of the graphblas. In *IPDPS Workshops*, pages 276–284. IEEE, 2019.

[69] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[70] Robert Clay Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[71] Semih Salihoglu and Jennifer Widom. Optimizing graph algorithms on Pregel-like systems. *Proceedings of the VLDB Endowment*, 7(7):577–588, 2014.

[72] Florian Bourse, Marc Lelarge, and Milan Vojnovic. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1456–1465, 2014.

[73] 9th DIMACS implementation challenge – shortest paths. http://www.dis.uniroma1.it/challenge9.

[74] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600, 2010.

[75] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In *European conference on principles of data mining and knowledge discovery*, pages 133–145. Springer, 2005.

[76] Paul Erdös, Alfréd Rényi, et al. On random graphs. *Publicationes mathematicae*, 6(26):290–297, 1959.

[77] Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, and Anish Das Sarma. Finding connected components in map-reduce in logarithmic rounds. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 50–61. IEEE, 2013.

[78] Chirag Jain, Patrick Flick, Tony Pan, Oded Green, and Srinivas Aluru. An adaptive parallel algorithm for computing connected components. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2428–2439, 2017.

[79] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[80] Andrew Chi-chih Yao. An $o(|e| \log\log|V|)$ algorithm for finding minimum spanning trees. *Information Processing Letters*, 4(1):21–23, 1975.

[81] David Cheriton and Robert Endre Tarjan. Finding minimum spanning trees. *SIAM Journal on Computing*, 5(4):724–742, 1976.

[82] Vitaly Osipov, Peter Sanders, and Johannes Singler. The filter-kruskal minimum spanning tree algorithm. In *2009 Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 52–61. SIAM, 2009.

[83] Bernard Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM (JACM)*, 47(6):1028–1047, 2000.

[84] Harold N Gabow, Zvi Galil, Thomas Spencer, and Robert E Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[85] Seth Pettie. Finding minimum spanning trees in o (m α (m, n)) time. 1999.

[86] David A Bader and Guojing Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *Journal of Parallel and Distributed Computing*, 66(11):1366–1378, 2006.

[87] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 12–25, 2011.

[88] Cristiano da Silva Sousa, Artur Mariano, and Alberto Proença. A generic and highly efficient parallel variant of boruvka's algorithm. In *2015 23rd Euromicro*

International Conference on Parallel, Distributed, and Network-Based Processing, pages 610–617. IEEE, 2015.

[89] Wei Zhou. *A practical scalable shared-memory parallel algorithm for computing minimum spanning trees*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2017.

[90] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, pages 1–12, 2013.

[91] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):1–39, 2015.

[92] Da Yan, Yingyi Bu, Yuanyuan Tian, Amol Deshpande, et al. Big graph analytics platforms. *Foundations and Trends® in Databases*, 7(1-2):1–195, 2017.

[93] Julian Shun, Laxman Dhulipala, and Guy Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 143–153, 2014.

[94] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[95] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.

[96] Md Mostofa Ali Patwary, Peder Refsnes, and Fredrik Manne. Multi-core spanning forest algorithms using the disjoint-set data structure. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 827–835. IEEE, 2012.

[97] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[98] Baruch Awerbuch and Yossi Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Transactions on Computers*, 10(C-36):1258–1263, 1987.

Acknowledgments

I would like to express my deepest appreciation to my advisor, Professor Zhenjiang Hu, for his expertise, insightful viewpoints, constructive suggestions and constant encouragements. Without his guidance and persistent help this thesis would not have been possible.

I would like to thank my committee members, Professor Kento Aida, Professor Kato Hiroyuki, Professor Kanae Tsushima and Professor Atsuhiro Takasu, for their insightful comments to improve this thesis.

I wish to acknowledge the support received from the staffs and Laboratory members in the National Institute of Informatics, Japan.

I wish to thank Professor Ariful Azad of Indiana University, with whom I have been co-authoring a paper, for leading me to the new field of high performance computing.

Finally, I would like to express the most heartfelt gratitude to my family members for their loving considerations and great confidence in me all through these years.