ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

IBM

LST
Laboratory for Software Technology

Carafe

High-Performance, In-Memory Graph Processing with RDMA

Master's Thesis

Clemens Lutz
ETH Zurich

October 2014

Advisors:

Animesh Trivedi
IBM Research, Zurich

Prof. Dr. Thomas R. Gross
Laboratory for Software Technology
ETH Zurich

Manager:

Dr. Thomas Weigold
IBM Research, Zurich

Abstract

Many real world problems operate on large graphs. Examples of these problems are reachability queries in social graphs like Facebook, determining ranks of web pages on the Internet, or calculating the shortest route in a city map. With the advent of Big Data, the size of such graphs is constantly increasing and can now comprise millions of vertices and billions of edges. Unsurprisingly, fast and efficient processing of large graphs has become an important class of problems in the modern data analytics stack. State-of-the-art graph processing systems divide and process graphs in a distributed setting using networked machines. Consequently, for a known I/O intensive workload like graph processing, the network I/O cost governs the overall performance. In order to reduce the network cost, these systems treat graph storage and computation as a joint problem and try to co-locate graph computation to where the graph data is stored, thus avoiding the expensive network I/O.

Inspired by the recent developments and the availability of high-performance network I/O (e.g. RDMA) in data center networks, in this work we step back and consider what the right abstraction for fast and efficient distributed graph processing is. We argue for decoupling and considering storage challenges separately from computation. In this thesis, we propose Carafe, a general-purpose graph storage and access framework that provides high-performance access to graph entities (vertices and edges) and associated computation states (vertex and edge properties) in a distributed setting. To illustrate the capabilities of Carafe, we have implemented online and offline (Pregel) graph processing applications. These systems perform well and our implementation of PageRank on a graph containing millions of vertices outperforms GraphLab and GraphX, two state-of-the-art systems, by a margin of 2.6× and 4.2×, respectively.

Acknowledgments

I would like to express my special appreciation and thanks to my supervisor, Animesh Trivedi. You have opened my eyes to the broader picture in research and patiently edged me towards the right direction when I lost sight of it, lapsing into details. Thank you for your open ear, technical and personal advice, time and effort.

My thanks extend to the wider team consisting of Dr. Patrick Stüdi, Dr. Bernard Metzler, and Jonas Pfefferle. You have included me as a team member and continuously provided council and guidance every step of the way. Jonas, thank you also for your loyal friendship and the numerous times you have listened to ideas, given input, and helped me to reflect on them.

I am much obliged to my mentor, Prof. Dr. Thomas R. Gross. Thank you for your openness towards writing the work externally from ETH Zurich at IBM Research, encouraging me to explore by asking the pivotal questions, all the while helping to find a fitting scope for this work. Without you, this thesis and all the knowledge and experience that came with it would have never happened.

Further, I would like to recognize my manager, Dr. Thomas Weigold, for his continued support throughout these past months.

Finally, my gratitude goes to my close friends and loved ones. You have bestowed upon me love and motivation, endured long phases of my absence from your sides, and subsequently welcomed me back into your midst. Thank you for always being there for me, unwaveringly, unconditionally.

Contents

1 Introduction

We are living in a data-centric world in which terabytes of data are routinely produced, stored, and processed every day. To capture trends, associations, and dependencies between data points in large data sets, researchers model them as graphs. Some real-world problems naturally occur as graph problems. For example, Facebook, Twitter, and LinkedIn all store friendships, interests, communities etc., as social graphs. Consider the background operations when LinkedIn suggests your friends' friends as people you may also know. The profile page of every person in the network shows a chain of people connecting them to you, as in Milgram's famous "small world" experiment. Similarly, entering a search term in Google results in millions of web pages listed by relevance. As a small but illustrative selection demonstrating large-scale use-cases, think of the problem statements behind them: the first finds friends-of-friends in your social graph by performing a breath- or depth-first search. The second ranks web pages on the Internet according to their hyperlink structure, and displays them against a search query in a sorted order by their ranks. All of these problems are intuitively expressed as graph operations. Consequently, fast and efficient processing on large graphs has become an important class of problem in the modern data analytics stack.

In a spirit similar to data-parallel frameworks, many distributed graph processing systems have been proposed. Examples of such systems include Pregel, GraphLab, GraphX, Trinity, and many more. These systems divide, store, and process graph data on a cluster of machines in a distributed setting. Distributed computation becomes a necessity to meet the storage, processing, and performance demands associated with large graphs. However, distributed large-scale graph processing is a challenging problem. Graph processing has become very context dependent and preprocessing or predicting results for *online* workloads is infeasible. It must be done in a fast, low-latency manner. An example of such workload is finding interesting news feeds that depend on your social graph, which includes structural information, such as your friends, as well as contextual information, such as the closeness of a friendship, mutual interests, permissions, and so forth. Another example is quickly calculating relevant advertisements based on a user's "like" graph. *Offline* graph analytics, in contrast, require high throughput to churn through vast data sets in their entirety. For instance, Google calculates PageRanks of all websites reachable on the Internet. Despite their differences, both paradigms have in common that graph processing typically imposes little computation per vertex, thus is I/O intensive with little or no predictability. The systems aspect of managing graph data in a distributed environment brings additional challenges associated with storage, parallelization, scheduling, ordering, and fault-tolerance.

Many of these systems treat graph storage and computation as a combined problem. This design choice gives them flexibility to co-locate computation and relevant graph objects on the same machine, thus to a large degree avoiding expensive network I/O. However, network communication cannot be avoided completely. It depends upon

(a) the graph structure that indicates how vertices are connected and communicate;
(b) the graph partitioning scheme that dictates which vertex communications are local (i.e. both sender and receiver vertices are located on the same machine) and which are remote;
(c) the graph computation itself that governs which nodes are participating in the communication.

As evident, to reduce the remote communicate cost one needs to come up with a sophisticated graph partitioning logic. However, real-world graphs, e.g. social graphs on Facebook or Twitter, are very densely connected and partitioning on such billion node graphs with equal load balancing

and minimal communication across partition boundaries is an NP-hard problem.

In this work, we step back and ask ourselves what the right abstraction for fast and efficient distributed graph computation is. We argue that graph storage challenges should be addressed separately from computation. Ergo, we decouple storage and computation concerns and optimize them individually. For example, which graph object access is local and which is remote should be addressed by the storage system, not computation. The computation system should focus on how to define and distribute workload independent of the data location, do global synchronization, implement algorithms and optimizations. In this thesis, we propose a general-purpose, high-performance graph storage and access framework that is not tailored towards any specific computational model. The goal of the storage framework is to provide high-performance access to graph entities (e.g. vertices, edges, properties), and associated computation states. *Carafe*, our implementation of said framework, achieves such performance by storing data in RStore. *RStore* is a high-performance, distributed, in-memory data store that uses high-performance RDMA operations to deliver high bandwidth, low-latency access to data stored in remote DRAM. RStore achieves this performance by building upon the separation philosophy of RDMA, and has explicit API calls to setup and access storage. Carafe's API pushes this separation philosophy further into graph processing. By using Carafe's API, graph computation frameworks can identify and create their necessary, distributed resources upfront, outside of performance critical sections, and later benefit from fast graph I/O operations during computation when performance really matters.

To demonstrate the flexibility and capabilities of Carafe, we have implemented algorithms directly on top of its API and developed a graph computation framework. The Carafe API supports general purpose, online, graph-exploration type computations such as Dijkstra's shortest path algorithm, breadth- and depth-first search. Though underlying graph data is stored in a distributed storage, the computation in this mode is not automatically distributed. The developed framework is distributed and implements the bulk-synchronous, graph-parallel Pregel computation model. In the Pregel model, we have implemented PageRank, shortest path based on a breadth-first search, and FloodEcho algorithms. Of these, we demonstrate Dijkstra's shortest path algorithm on the Carafe API, and PageRank on the Pregel model.

Our contributions are the following:

- We propose to decouple graph storage from computation. We demonstrate the feasibility of our approach by designing and building Carafe, our proposed system.
- We design and implement a system API that is built upon the separation philosophy of RDMA. It delivers high-performance graph access during computation by letting applications identify and create graph resources upfront.
- We evaluate Carafe and compare the results with state-of-the-art frameworks in graph processing.
- We analyze Carafe's behavior under conditions of constrained data locality and individual message passing; unlike others, the system is not inherently designed to minimize I/O.

The thesis is organized as follows. First, in chapter 2, we describe RDMA, explain the RStore framework, and provide an overview of graph- and graph-processing-related topics. In chapter 3 we then discuss the design and architecture of Carafe and proceed with detailing the implementation thereof in chapter 4. Chapters 5 and 6 discuss the same topics for our Pregel framework. In chapter 7 we provide an in-depth evaluation of our framework and compare our results to other systems representing current state of the art. Next, chapter 8 positions our solution in the context of prior work. Chapter 9 lists future work. Finally, in chapter 10 we summarize our work, discuss and draw conclusions from it.

2 Background

At its core Carafe is a distributed graph storage and processing system. It uses RStore, a distributed, in-memory data store built using RDMA. In this chapter we first present background on RDMA technology and how RStore leverages it to deliver high-performance graph data access to Carafe. Subsequently we introduce relevant graph theory and graph processing models. To close the gap between theory and models, we also discuss techniques to store and process graphs, and their various trade-offs.

2.1 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) is a high-throughput, low-latency networking technology in which connected network end-point applications directly access memory buffers for data transferal. It follows a separation philosophy in which remote data access is decoupled from the connection setup and control. An application that wants to access remote data first goes through a connection and resource (buffers, queues, etc.) setup phase. Then, for data access, the application interacts directly with an RDMA-capable network interface (*RNIC*) on a fast data path, by-passing the operating system and CPU. This separation of control and data is key to eliminating unnecessary operating system and CPU involvement, such as data copies, interrupts, multiplexing, and scheduling. Elimination of these results in true zero-copy, high-performance data access. In this thesis we put focus on preserving RDMA's separation philosophy and maintaining a zero-copy stack.

The "RDMA Protocol Verbs Specification"[1] describes an abstract API which defines the RNIC's behavior and its interaction with software. An implementation will then expose a concrete API to applications. We describe the Verbs functionality relevant for this document.

2.1.1 RDMA Verbs

An RNIC receives instructions encoded as a set of operations defined by RDMA. Operations are placed as *work requests (WR)* on a *work queue (WQ)*, which can be either a send queue or a receive queue. Together they form a *queue pair (QP)*. A *completion queue (CQ)* is then bound to one or more work queues. The RNIC posts *work completions (WC)*, which are notifications of completed work requests and state associated with WCs (i.e. success, error). Work requests operate on a *memory region (MR)*, which consists of one or more pages of physical main memory. A memory region must be registered at the RNIC before any WRs may reference it. At registration time the contiguous virtual memory buffer is translated to the underlying physical memory pages, and during operations the RNIC accesses physical memory directly. As this translation is done only once at registration time, the pages of a MR must be "pinned" by the operating system so that the virtual address to physical address mapping remains fixed and the operating system does not e.g. swap them to disk. A *steering tag (STag)* identifies and grants access to a registered memory region. Together with a *tagged offset (TO)*, it specifies a memory location, to which network I/O accesses have byte-level granularity. QPs and MRs are associated to one another via a *protection domain (PD)*. The RNIC enforces protection of memory regions for all accesses passing through it by validating the STag and matching the protection domain before a work request is executed on the memory region.

There are two fundamental types of operations: (a) one-sided and (b) two-sided operations.

One-sided operations allow two peers to communicate without involving the CPU of the remote peer on the data path — the requests are handled entirely by the RNIC. Hence, the RNIC must be capable of offloading the entire network stack. There exist two one-sided operations, *read* and *write*. These one-sided operations require the remote peer to register a memory region and advertise a STag to the local peer. Read and write operations then access memory locations within the memory region with this STag and a tagged offset.

Two-sided operations also by-pass the CPU for data accesses, but require both peers to jointly interact with message-passing semantics. For all types of operations, the caller may optionally request a *completion notification* to be issued by the RNIC once the operation is completed, indicating that a new work completion is located on the completion queue. However, completion notifications are in effect hardware interrupts, which require CPU and operating system involvement and thus incur additional latency. Like the one-sided operations, there exist two two-sided operations, *send* and *receive*. The receiver places a receive work request on the receive queue with a memory buffer large enough to place the incoming data. The sender places a send work request on the send queue, which points to a send buffer. If the receiver is not yet ready, the sender receives an error on the completion queue and must try again. Unlike with one-sided operations, the receiver is not required to advertise an STag.

2.1.2 Implementations

The OpenFabrics Alliance's "OpenFabrics Enterprise Distribution" (OFED) is a widely-used, concrete API implementation of the RDMA Verbs specification and is made available for Linux and Microsoft Windows. The OFED stack supports the three most common, RDMA-capable networking technologies: Infiniband, iWARP for Ethernet, and RDMA over Converged Ethernet (RoCE). In this work we will focus on iWARP. However, we do not have any transport-specific dependencies.

2.1.3 iWARP Atomicity and Ordering Guarantees

iWARP is a standardized[2] RDMA protocol for IP, using either TCP, SCTP, or UDP as transport. Although multiple lower-layer protocols are supported, the most commonly used is TCP. TCP itself guarantees reliable data transport. iWARP then guarantees sequential arrival ordering of work requests within a work queue. Further, operations on memory have byte granularity, and byte accesses are atomic. However, before the release of RFC 7306, iWARP did not support atomic operations such as fetch-and-add or compare-and-swap, which are helpful for implementing concurrent data structures and non-trivial synchronization mechanisms. Also, iWARP, unlike Infiniband, does not provide any guarantees on the data placement order, meaning that peeking into a memory buffer during an ongoing data transfer results in undefined behavior[2, p. 38]. Both RStore, our distributed data store, and Carafe are designed keeping these limitations in mind.

2.2 RStore

Carafe uses *RStore*[3] to store and access graph data. RStore is a general-purpose, distributed, in-memory data store. Applications can develop, store, and share distributed data structures, schemas, tables, data-blobs etc. in it. Similar to distributed, shared memory, RStore does not interpret or impose any structure on the stored data. This avoids the overhead of the implicit I/O operation synchronization associated with data structures. However, unlike distributed shared memory, RStore is not designed to provide a cache-coherent memory extension to run applications transparently on a cluster. The authors state that RStore is built on two key design principles: (a) "decouple resource allocation from its abstraction binding", in other words manipulating, multiplexing, and recycling expensive memory and network resources independently

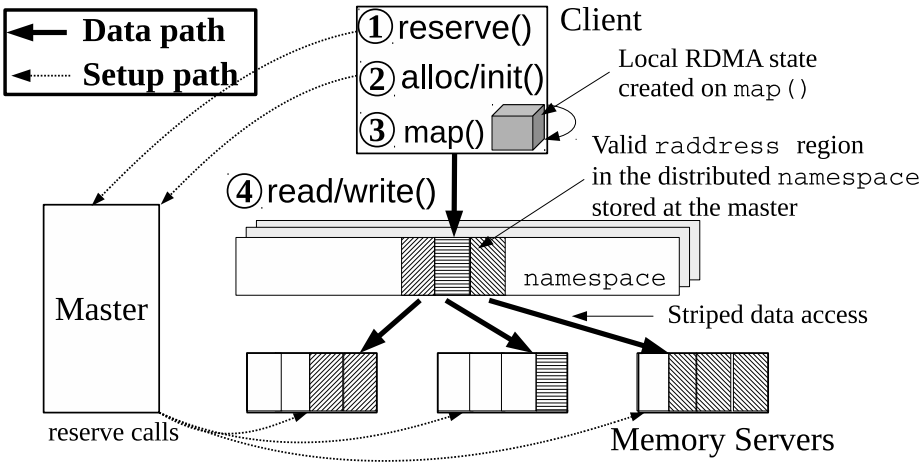


Figure 2.1: Design and Components of RStore.

of the provided storage abstraction, and (b) "keep the I/O path thin and fast", where on the I/O path data is striped across multiple servers. In this section, we give an overview of RStore's design and describe its inner workings relevant to our work.

2.2.1 Components

RStore is comprised of three main components: one master, one or more memory servers, and an arbitrary number of clients (in the following chapters of this document, we will refer to these as the *RStore master*, *RStore servers*, and, if referring to the RStore component embedded within a client application, *RStore clients*). The master is a logically centralized entity for arbitration of resources and storage of system-wide metadata. Data is distributed and stored on the DRAMs of participating memory servers. Their primary responsibility is to allocate and prepare DRAM buffers to be accessed by RStore clients. Clients are applications that interface with the RStore API made available as a library. Carafe is an RStore application that uses its API to store graph data in it. Figure 2.1 captures the interaction among these components.

2.2.2 Abstractions and API

RStore provides a flat, 64-bit byte-addressable storage abstraction called *RStore namespace (NS)* or just *namespace*. A namespace is associated with a globally unique identifier string, i.e. its name, which is given by the client when creating a namespace. A new namespace can be created by calling `create()` on an *RStoreMaster* object. Namespaces are globally visible and accessible. An existing namespace is retrieved by calling `get_namespace()` with a name. A namespace object, as returned by the `get_namespace()` call, can be made ready for storage by calling `join_namespace()` on it. RStore clients can create, join and allocate storage capacity in multiple namespaces. An allocated storage region is identified by a globally visible (through the master) ⟨address, length⟩ tuple, called an *RStore address (raddress)* object. Allocated raddress regions are physically served by DRAM from multiple memory servers for storage. Multiple clients can access the same raddress region. Carafe uses separate namespaces to store graph data (e.g. vertices, adjacency lists) and computational state. Examples of latter are communication messages or synchronization states. These namespaces are identified and accessed by multiple clients in a distributed way using their meaningful string identifiers.

RStore's API provides a familiar memory-mapped I/O framework (`mmap()` and friends), through which applications can allocate, map, read, and write remote memory at byte granularity. RStore adheres to the separation philosophy of RDMA and provides separate API calls to

setup and access storage resources in a distributed setting. This separation enables RStore to hide the low-level complexities of the RDMA Verbs interface behind a higher-level API, while retaining and passing through RDMA's high performance to applications. Three explicit API calls, namely `reserve()`, `alloc()`, and `map()`, constitute the distributed control path in RStore. They allocate resources and create states on memory servers, the master, and clients, respectively. Calling `reserve()` with a size value, given in bytes, on the namespace object instructs the master to prepare memory on the servers. The `alloc()` call, which also takes a size value, stitches together prepared memory on the servers and returns it as an raddress object. Finally, the remote memory is accessed by first calling `map()` on the valid raddress, and then subsequently calling the data operations `read()` and `write()` on it. Read and write calls are part of the fast data path. For calls involving resource allocation, either in terms of memory or network resources, converse calls exist to free the allocated resources. Carafe's storage API, which respects the separation philosophy, is built upon the RStore API and abstractions. Carafe's API provides high-level calls for common graph access patterns, such as sequential and random vertex access or linear adjacency list scans, while implicitly managing resources in RStore in an efficient way.

2.2.3 Distributed Memory Management

DRAM storage is managed internally in a granularity of *chunks*. Each chunk physically consists of one or more memory pages located on a memory server. The master tracks free chunks in a free chunk list, allocated chunks in a per-namespace allocated chunk list. A list of free chunks, chunks allocated to raddress regions, their access permissions, mapping types, active clients, and namespaces created, is maintained as system metadata at the master. Clients can allocate, map and access raddress regions of any size in any namespace. Hence, a namespace contains multiple raddress regions, which themselves consist of one or more chunks with their location-to-chunk mappings.

Storing and accessing data in RStore are multi-step processes. First, an application must reserve enough DRAM capacity by calling `reserve()` on a namespace object. The call issues an RPC from the client to the master, which checks the free chunk list for the requested memory size. If found, the call returns immediately. Otherwise, the master chooses a number of memory servers and issues RPCs to them asking for memory allocations. Upon receiving a `reserve()` RPC call from the master, memory servers allocate and register chunks of DRAM to an RDMA device and communicate RDMA credentials — namely STags, addresses and lengths — back to the master. The master adds these chunk credentials, together with the memory server's IP, to its free chunk list. The current implementation uses a primitive round-robin policy to uniformly distribute the load.

Second, an `alloc()` call issues another RPC to the master, which stitches together chunks from the free list to create a new contiguous memory region in the namespace. This globally visible mapping between an raddress region and memory chunks is created and stored at the master. The newly created raddress region together with its chunk locations are returned to the client as the result of the `alloc()` RPC. A previously allocated valid raddress region object can be initialized by calling `initialize_address()` with an address and length on it. Upon receiving an initialization RPC, the master looks up and returns the raddress-to-chunks and chunk-to-location mappings together with other metadata to the client as a return value for the RPC. This return value is used to initialize the raddress object on the client side. The master additionally stores a reference counter for each namespace to avoid freeing the mapped chunks while these are still in use. Both `alloc()` and `initialize_address()` atomically increment the reference counter, destruction of an raddress object atomically decrements it. Atomicity implies that these operations are globally serialized when called on the same namespace.

Lastly, a valid raddress object requires a local mapping, created by a `map()` call, before the I/O operations. Similarly to the `mmap()` call, a local mapping returns a local DRAM address,

which is made RDMA-ready by the RStore client library. Clients use this address for staging and modifying their data. There is an equivalent `unmap()` call. RStore performs resource caching and sharing to hide the high control setup cost for repeated accesses. The RStore client library manages and caches RDMA-ready memory using a user space buddy allocator. RDMA objects, e.g. connections to memory servers, event channels, or I/O queues, are shared between multiple mappings.

The `reserve()` and `alloc()` call separation decouples the former call's control of memory and network system resources from the latter call's binding of resources to the abstraction. That is, setting up the RDMA data path via the RDMA control path and allocating memory are no longer directly coupled to the mapping of an raddress to chunks and their placement within a namespace. This allows RStore to pre-allocate server memory, and reuse the memory after it is deallocated by the client. Likewise, separating the `initialize_address()` and `map()` calls means that the metadata necessary for accessing remote memory regions and the memory buffers needed to represent the remote memory locally are independent of one another — the raddress metadata can be cached and used multiple times to map the segment it references, while one and the same local memory buffer can be reused for a new mapping once the previous one has been unmapped. Furthermore, the `map()` call, and by extension the `unmap()` call, are completely local. If an application moves all calls to `initialize_address()` out of its fast path, the fast path will involve no network operations whatsoever. Additionally, once the application has reached its peak memory footprint, mapping predominantly reuses buffers, avoiding the costs of buffer creation and destruction. The efficient management of resources, as described above, is enabled by the key design principles of RStore, which are reflected in its API. The design of the operations beneath the API ties together all of these factors. In a similar spirit, Carafe internally caches RStore objects and associated metadata to avoid network round trips and minimize the access latencies to graph objects.

2.2.4 I/O Operations

A mapped raddress object supports `read()` and `write()` calls within the mapped memory region. These calls constitute the fast and thin I/O path of RStore. They do not involve any resource allocations anywhere in the system and are translated to one-sided RDMA read and write operations for zero-copy network transfers. These properties are key to achieving high performance in a low-latency, high-bandwidth environment. The byte-granular nature of I/O operations in RStore enables Carafe to calculate, index, and access graph data, such as adjacency list access for any random vertex, with utmost efficiency.

I/O operations in RStore do not provide any atomic or global ordering guarantees (as in iWARP, see section 2.1.3). Concurrent graph I/O operations in Carafe built on them also do not provide any global ordering guarantees. High-level frameworks on Carafe, for example C-Pregel, are built by exploiting the fact that accesses in a graph-parallel framework are split and ordered around a global synchronous barrier (see section 5).

2.3 Graph Theory

A graph is a mathematical concept used to concisely describe how a set of objects are interconnected. A graph is formed out of two types of components, vertices and edges; each object is represented by a *vertex*, and each connection between two objects is represented by an *edge*. More formally, a graph consists of two finite[1] sets, one of vertices and one of edges, denoted as $G = (V, E)$. The vertices $u, v \in V$ at each end of an edge are called the edge's *endpoints*, and are designated *adjacent* to one another (alternatively: u and v are *neighbors*). If not mentioned otherwise explicitly, we assume that adjacent vertices are connected by precisely one edge. Whereas

[1] We do not consider infinite graphs in this document

a vertex may be an endpoint of an arbitrary number of edges, every edge has exactly two endpoints. Going from one endpoint of an edge to the other endpoint is to *traverse* an edge. If an edge $e_d \in E$ is traversable in only one direction, thus one endpoint is its *head* and the second its *tail*, the edge is a *directed edge* and is denoted as the ordered pair $e_d = (u, v)$ of its endpoints. Otherwise, if the edge $e_u \in E$ is traversable in both directions, the edge is an *undirected edge* and is denoted as the unordered pair $e_u = \{u, v\}$ of its endpoints. The endpoints of a directed edge have different views of it: from the perspective of e_d's head u, it is an *outedge*; but from the perspective of its tail v, it is an *inedge*. By definition, all edges in a graph are of the same type, either directed or undirected, and hence the graph is then a *directed graph (digraph)* or an *undirected graph*, respectively. The number of endpoints a vertex v represents is denoted in the vertex's *degree* $deg(v)$. In a digraph, the degree is further separated into the outdegree $deg^+(v)$ and the indegree $deg^-(v)$, for outedges and inedges, respectively.

2.3.1 Power-Law Graphs

Graphs are often categorized into graph families by common properties, e.g. the degree distribution of their vertices. In this document, we focus on one such family: power-law graphs.

In a *power-law graph (power graph)*, the degrees of vertices follow a power-law probability distribution. As an illustration we consider people and their friends in an online social network. There are many people (vertices) with a small number of friends (degrees). The higher the number of friends, the lower the number of people with that many friends. Only very few people, for example celebrities, have an extraordinarily large number of friends.

The probability mass function of the power-law distribution is defined[4] as:

$$p(x) \propto x^{-\alpha}$$

where x is a vertex degree, $p(x)$ is the probability of a vertex having that degree, and $\alpha > 0$ is a constant deciding how fast the probability declines with growing x.

Power-law graphs are relevant as social, web, and many other real-world graphs follow a power-law degree distribution[5, 6, 4].

2.4 Graph Data Structures

When storing large graphs in a data structure, we pursue two goals: minimizing space usage and minimizing access time. Many graphs, such as power-law graphs, are sparsely populated with edges — storing them quickly becomes inefficient if the format does not adapt to sparsity. Simultaneously, access to a particular element within a graph should also be efficient; ideally, elements are indexed.

As so often, the challenge is to unify both goals in a single data structure. There are a range of fundamental storage formats with different properties and trade-offs to evaluate: the storage space complexity, and the time complexities of iterating the neighbors of a vertex, querying for vertex adjacency, adding and removing vertices or edges. The adjacency query assumes that the source and destination vertices are known, but the edge itself is unknown. Note that the following descriptions assume arrays as underlying storage; for other data structures the time and space complexities may be different.

2.4.1 Incident Matrix

The *incident matrix* is an inefficient approach to storing a graph. Nevertheless, it is a good demonstration of what *not* to do.

An undirected graph is represented as a two-dimensional matrix with $|V|$ rows and $|E|$ columns. The rows represent vertices, the columns represent edges. Each matrix entry stores

a Boolean value, which is `true` if the vertex is an endpoint of the edge (called *incident* in this context), and `false` otherwise. Directed edges can be represented by adding a sign flag to each entry for designating an incoming or outgoing edge, making the incident matrix *oriented*.

An incident matrix thus uses at least $\frac{|V|\cdot|E|}{8}$ bytes of space. An undirected edge is incident to only two vertices, meaning this structure wastes at least $\frac{|V|-2}{8}$ bytes for every edge stored.

Neighbor iteration involves iterating the vertex's row to find all non-zero entries. Then the column of each non-zero value is iterated to find the second non-zero entry of the column, which specifies the neighbor. All of this is in $O(|V| \cdot |E|)$ time. An adjacency query is effectively a neighbor iteration which stops when the queried neighbor is found. Adding a vertex uses $\Omega(|V|)$, adding an edge $\Omega(|E|)$ time in the best case — that is, when expanding the matrix can be done by simply appending a row or column. This is not always feasible, and the whole matrix must be reallocated and copied. Thus, the worst case for add (and remove) operations is $O(|V| \cdot |E|)$ time.

2.4.2 Edge list

Improving on the incident matrix is the *edge list*. Instead of taking a vertex-oriented approach, it views the graph as a list of edges in no particular order. Each edge is explicitly represented as a ⟨source vertex, destination vertex⟩ tuple and can be either seen as directed or undirected. Note that an edge list cannot store vertices without edges, although this limitation can be worked around by introducing a placeholder (or "dummy") vertex.

The space required to store a graph is in the order of $O(|E|)$. Adding an edge is trivial and takes $O(1)$ (amortized) time. Neighbor iteration, neighbor querying, and removing an edge all involve iterating through the edge list, taking $O(|E|)$ time. Removing an edge means to first find it and then to fill the gap it leaves, totaling in $O(|E|)$ time. Adding and removing a vertex are equivalent to adding or removing an edge, respectively.

2.4.3 Adjacency Matrix

The *adjacency matrix* is another way of improving upon the incident matrix. The adjacency matrix is a two-dimensional, square matrix with $|V|$ rows and columns. The rows represent source vertices and the columns destination vertices. Each matrix entry stores a Boolean value, which is `true` if the two vertices are adjacent to one another, i.e. there is an edge connecting them. Alternatively, the Boolean value can be replaced by the edge's weight, where a zero weight edge stands for "not adjacent". When storing an undirected graph the adjacency matrix is symmetric — storing it as a triangular matrix suffices.

The space required to store a graph is in the order of $O(|V|^2)$. This is especially efficient for dense graphs, but quickly becomes inefficient for sparse graphs.

Neighbor iteration involves iterating a vertex's row to find all non-zero columns, thus $O(|V|)$ time. An adjacency query is simply a lookup in the matrix and is in $O(1)$ time. Adding or removing a vertex is similar to the incident matrix's case and is again in the order of the matrix's size, $O(|V|^2)$. However, adding or removing an edge is cheap, involving only a bit flip and thus $O(1)$ time.

2.4.4 Compressed Row Storage

The Achilles heel of the adjacency matrix is storing sparse graphs, as it converts sparse graphs into sparse matrices. Storing sparse matrices is a well-known problem and many special cases have storage formats tailored to them. One of these is the *compressed row storage format (CRS)*. With it storing sparse adjacency matrices becomes more efficient.

A CRS matrix consists of three vectors: the first vector stores non-zero entries representing adjacencies, the second contains the column index for each of these entries, and the third rep-

resents all rows and contains offsets into the first two vectors. Therefore, row index n holds the index of the first non-zero entry of row n, at which position that entry's column index is stored in the column vector. Put another way, a CRS matrix is an array of pointers to arrays containing ⟨entry, column index⟩ tuples.

Consider an adjacency matrix stored in CRS format. Space complexity is now in the order of $O(|V| + |E|)$, which is much more efficient for sparse graphs. It still holds the $O(|V|^2)$ upper bound from the adjacency matrix, but now includes some constant overhead to store indices. Also, accessing entries include an overhead of one indirection, which is in constant time. Other time and space properties are the same as with adjacency lists, described below.

2.4.5 Adjacency List

An adjacency matrix stored in CRS format carries unnecessary baggage; the existence of an adjacency is already given when storing its column index, storing an additional Boolean is superfluous. For this reason, the CRS format is reduced to an *adjacency list*. In the adjacency list data structure a graph is represented as a vector of vertices, where each vertex includes a list of other vertices adjacent to it. It naturally stores the outedges of a directed graph. To store an undirected graph, each edge must be split into two directed edges, essentially converting the undirected graph to a directed graph. Abstracting the graph format from the matrix-oriented view has another advantage: the underlying arrays can be exchanged for another type of data structure with different properties.

Retaining the assumption of arrays underlying the adjacency list, storage complexity is in the order of $O(|V| + |E|)$.

Neighbor iteration is trivial; finding it involves a lookup of the vertex followed by a jump to the neighbor list, for a total of $O(|V|)$ time. As the neighbor list is contiguous, this operation is very efficient. An adjacency query involves iterating neighbors until the one searched for is found; this is again in $O(|V|)$ time. Adding or removing a vertex means to resize the vertex array, which takes $\Omega(1)$ time in the best case and $O(|V|)$ time in the worst case. Adding or removing an edge means to resize the adjacency list's array. If all adjacency lists are stored in one array, this takes between $\Omega(1)$ and $O(|E|)$ time. If each adjacency list is stored in a separate array, this takes between $\Omega(1)$ and $O(|V|)$ time.

Considering the trade-offs involved, our design of Carafe stores graphs in the adjacency list format. However, Carafe's API is independent of the underlying graph data structure.

2.5 Graph Algorithms

To evaluate the Carafe framework, we have targeted two specific algorithms, sequential Dijkstra and PageRank on the Pregel model (see section 2.6.1).

2.5.1 Dijkstra's Algorithm

Dijkstra's algorithm (Dijkstra)[7] solves the *single source shortest path (SSSP)* problem. The single source shortest path is defined in Dijkstra's original article as "*finding the path of minimum total length between two given nodes (vertices) P and Q*". This definition assumes that the positions of the two given vertices in the graph are known and that finding one out of potentially many shortest paths is sufficient.

On a high level, the algorithm uses the implication that, if a vertex R is on the minimal path between P and Q, then the minimal path from P to R is also known. Thus the algorithm performs a greedy search for all minimal paths from P until Q is reached.

Let the distance of a vertex be the length of its minimal path from P, and let the vertex on the minimal path immediately preceding a vertex R be R's parent. Then Dijkstra's algorithm works as follows:

1. Initialize all vertices in the graph to have distance ∞, but P to have distance 0. Further, mark all vertices as unvisited, and place P into a set and name it the active set.
2. Remove the vertex with smallest distance from the active set and let it be the current vertex.
3. If the current vertex is Q then stop.
4. If it is marked as visited, repeat from step 2. Else, mark the current vertex as visited.
5. For each neighbor N, let d be the sum of the current vertex's distance and the distance between the current vertex and N.
6. If $d > N$'s distance, inspect the next neighbor. Else, let d be N's new distance, let the current vertex be N's parent. If N is not in the active set, add N to the active set. Else, update N's position in the active set to match its decreased distance.
7. Repeat from step 2.

The operations of adding a vertex to, removing the vertex with smallest distance from, and updating the position within the active set make up a significant portion of the algorithm's time complexity. If implemented with a minimum, binary heap, and assuming the graph is stored in an adjacency list, the algorithm's worst-case time complexity is $O((|E|+|V|)\log(|V|))$[8, p. 199].

The algorithm does not directly output the minimal path between P and Q, but rather the set of minimal paths from P to all visited vertices. Finding the minimal path to Q is most efficiently done by backtracking from Q, that is following the parent references from Q until P is found. Backtracking is in $O(n)$ time, where n is the minimal path's length.

2.5.2 PageRank

PageRank is an algorithm to calculate a homonymous metric for measuring the relative importance of a vertex within a graph. Intuitively, the importance of a vertex v is defined as the likelihood that a random sequence of edge traversals originating from a randomly chosen start vertex will come across v. An inedge from a highly-ranked vertex carries high value, unless the highly-ranked vertex has many outedges.

PageRank formalizes this intuition as[9]:

$$R'(u) = c \sum_{v \in B_u} \frac{R'(v)}{N_v} + cE(u)$$

where u is a vertex, R' is a vector of PageRanks, B_u the set of vertices with outedges to u, N_v the number of outedges of v, c a normalization factor, and E a vector of constants. From the formal definition it follows that PageRank is an iterative algorithm converging to a solution.

2.6 Graph Processing Models

Graph processing models provide structural and operational primitives. The structural primitives abstract data into a graph. The operational primitives access and modify the graph structure or data associated with the structure. Finally, graph processing models define an execution order that facilitates the combination of these operational primitives into meaningful algorithms or queries on the graph.

There are two perpendicular ways of categorizing graph processing models. The first splits models by the type of queries and algorithms they handle and the source of mutability of the data, i.e. internal or external to the model's computation. The second splits by the model's view on the graph; the granularity with which a graph is passed as input to a computation is defined by the view. The view is either *graph-*, *vertex-*, or *edge-oriented*. If the view is graph-oriented, the entire graph is passed as input to the computation. As such, it has a global view of the graph and can access any component at any time. In contrast, if the view is vertex- or edge-oriented, the computation is divided into units and only a single component at a time is passed to a computational unit as input. In such a model, the model defines a mechanism for units to communicate, either based on message passing or shared memory and an (partial) order in which they are executed.

Models are further differentiated as *online* and *offline* models. Online models handle interactive queries on an often live, constantly changing graph. They come with query-oriented operational primitives which are suitable for substituting or building high-level, relational queries. As they operate on live data, their main challenges are providing consistency guarantees and low-latency. Furthermore, online models are often graph-oriented as to allow computations to operate on the whole dataset. Offline models cater towards batch processing of mostly static data, in which all changes to the data originate from the computation being executed. They include operational primitives targeted at distributed graph algorithms operating on graph components individually, and thus have a vertex- or edge-oriented view. They are also called *graph parallel* models due to their highly concurrent nature and their primary challenges are distributed synchronization and high throughput.

Offline models are subdivided into *synchronous* and *asynchronous* processing models, based on their communication model. The synchronous model breaks graph-parallel computation into discrete, global iterations. In each iteration, every computational unit is executed once, during which time it may have outgoing communications to other units, which are passed as input in a defined, subsequent iteration. The iteration is concluded with a barrier synchronization across all units. Because the order of unit execution and timing of inputs are strictly defined, synchronous models are deterministic. Additionally, because of the global barrier between the production and consumption of all communications and the single barrier in the model, a synchronous model is deadlock- and data-race-free[10]. In contrast, the asynchronous model is entirely event driven. An implementation-defined system scheduler decides when to execute a computational unit. A unit is only scheduled for execution if it receives a communication as new input, processes it, and then returns control to the system scheduler. The scheduler then synchronizes event production and consumption. Varying scheduler implementations and real-world factors such as variance in network latencies induce non-determinism in the execution of these models.

Although every synchronous problem can be reduced to an asynchronous one with the help of a synchronizer, and thus the asynchronous model is more general than the synchronous one, synchronizers come at the expense of time and message complexity. Depending on the fundamental assumptions made by an algorithm and the properties, e.g. the convergence rate it exhibits when run in a particular model, choosing the synchronous or asynchronous model over the other can have a big impact on runtime in practice[11].

2.6.1 Pregel Model

The Pregel model is a synchronous, vertex-oriented graph processing model with message passing communication semantics and is inspired by the *Bulk Synchronous Parallel model (BSP)*[12]. It was introduced by Google's "Pregel" graph processing system[10] and has seen wide-spread recognition in graph processing research.

In the Pregel model, the sequence of iterations of the synchronous computation are named *supersteps*. Computation is seen from the perspective of a vertex, thus, "a vertex computes" means that the system calls a method on the vertex. Each vertex can store and has access to

private, mutable storage. Further, each vertex can be in one of two states, *active* or *inactive*. Initially, all vertices are in active state. In each superstep k, a user-defined method is called individually on each active vertex in the graph, concurrently. When executed, a vertex receives the messages sent to it from the previous superstep $k-1$, and it sends messages to other vertices along its outgoing edges. These sent messages are delivered in the next superstep, $k+1$. During method execution, a vertex can set itself to inactive state by explicitly calling halt. A vertex is reactivated if a message destined for it arrives, in which case it must explicitly deactivate itself again. If, and only if, all vertices are simultaneously in inactive state and no messages are in transit, the system terminates. The output of the computation is the set of values each vertex explicitly defines as output, conjuncted over all vertices.

Pregel API The centerpiece of the Pregel API is the abstract `Vertex` class and its virtual `compute()` method. A Pregel application is implemented by subclassing `Vertex` and overriding `compute()`, which is called in every superstep. From within `compute()`, the current superstep number can be obtained by calling `superstep()` on the current object. The private vertex storage is readable with the `get_value()` and mutable with the `get_mutable_value()` methods.

Incoming messages are passed to the `compute()` method via an iterator given as argument. Though the order of the messages within the iterator is unspecified, the iterator is guaranteed not to contain duplicate messages and messages are guaranteed to arrive. Outgoing messages are sent by calling a `send_message_to()` method on the current object, where each message must specify one destination, given as a vertex ID. The ID of the current vertex can be obtained by calling `get_id()`, the IDs of neighbors from `get_outedge_iterator()`. As it is a common pattern to send the same message to all neighbors, the API offers an additional `send_message_to_all_neighbors()` method.

To reduce the overhead of messaging, the application can optionally implement a *combiner*. A combiner is an optimization for algorithms which use a commutative and associative aggregate function (e.g. sum, mean, count, max, etc.) on the values passed as messages. It combines values in multiple messages to one with the aggregate function. This reduces the number of messages, saving both memory to store messages and network bandwidth to transmit them. A combiner is implemented by subclassing the abstract `Combiner` class and overriding its virtual `combine()` method.

In addition to the described functionality, the Pregel API supports global aggregate functions and structural graph mutation, which are not discussed in further detail in this document.

3 Carafe Design

3.1 Vision and Design Goals

Graph processing is an I/O intensive application with very little computation done per graph component. Low latency and high throughput access to the graph are key for high performance. However, large graphs necessitate distributed storage of the dataset, making high I/O performance challenging. The high-performance RStore remote data store enables fast remote data access, but requires the data and control path separation to be passed up through the software stack to the application to reap the data store's full performance gains. The separation limits the design space, should the graph processing framework aim to take advantage of it — a further design challenge.

Given these constraints, we envision Carafe to operate in a dense, rack-scale deployment with many cores and multiple high-speed, RDMA-capable network links per machine. The dense deployment minimizes the latency costs associated with distance, network switches, and routers. Density also fosters tighter integration of a machine cluster with multiple network links per machine, scaling throughput beyond the capabilities of a single link. Finally, applications are distributed and many cores give them further opportunity to parallelize the processing of concurrent portions in their workloads.

Our design goals for Carafe are:

General, reusable abstraction Distributed, high-performance graph storage is useful for both online and offline graph processing applications — they should focus on application-specific logic and not be burdened by details on graph storage. Our abstraction should be flexible to suit the requirements of both graph processing models by providing the high-level, unbiased primitives for creating, reading, and modifying a graph and components thereof.

Distributed, uniform graph access Graph processing applications are distributed and parallel, but for optimal performance they require input to be evenly distributable among processing units along with minimal interdependencies to avoid network I/O. Unfortunately, partitioning a graph to meet these constraints is an NP-hard problem. Instead, the system should provide uniform access to the entire graph independent of the location the request is coming from, making graph partitioning unnecessary.

Data and control path separation To extract maximum performance from the hardware, Carafe must endorse the data and control path separation philosophy of RStore. The API should expose the separation to the application, and resource allocation within the system should be pushed onto the control path. The data path should remain fast and thin.

Data-copy and indirection avoidance Data copies and indirections inhibit performance and should be avoided wherever possible. Accesses to both application data and internal metadata should involve zero copies, placing data and accessing it directly instead.

To limit the scope of our work, we constrain Carafe to graphs with immutable structure. That is, vertices and edges cannot be added or removed from the graph. Supporting all structural graph operations would be difficult, since the fundamental graph data structures all impose penalties on at least one such operation. In large graphs these penalties quickly become expensive, necessitating more elaborate structures which in turn impose penalties on simple accesses. In this work, we focus our efforts on fast computation and data access.

Class	Method Call Signature		Description
CarafeGraph	`create(namespace, graph_name, |V|, |E|)`		Creates a new graph with the given size in RStore
	`load(namespace, graph_rstore_address)`		Loads an existing graph from RStore
	`unload()`		Unloads the graph locally
	`destroy()`		Destroys the graph in RStore
	`get_vertex_by_id(vID, vertex_handle&)`		Initializes the vertex handle to the given vertex ID
	`iter_from_to(begin_vID, end_vID)`		Returns a mutable vertex iterator pair from begin ID to end ID
	`citer_from_to(begin_vID, end_vID)`		Returns a constant vertex iterator pair from begin ID to end ID
VertexHandle	`get_id()`		Returns the vertex's ID
	`get_name()`		Returns the vertex's name
	`get_properties()`		Returns a reference to the vertex's properties
	`cbegin()`		Returns a constant adjacency list iterator at beginning
	`cend()`		Returns a constant adjacency list iterator at end
	`map()`	only for random access	Map vertex into local memory
	`unmap()`		Unmap vertex
	`read()`		Read vertex from RStore
	`write()`		Write vertex to RStore

Table 3.1: Carafe API

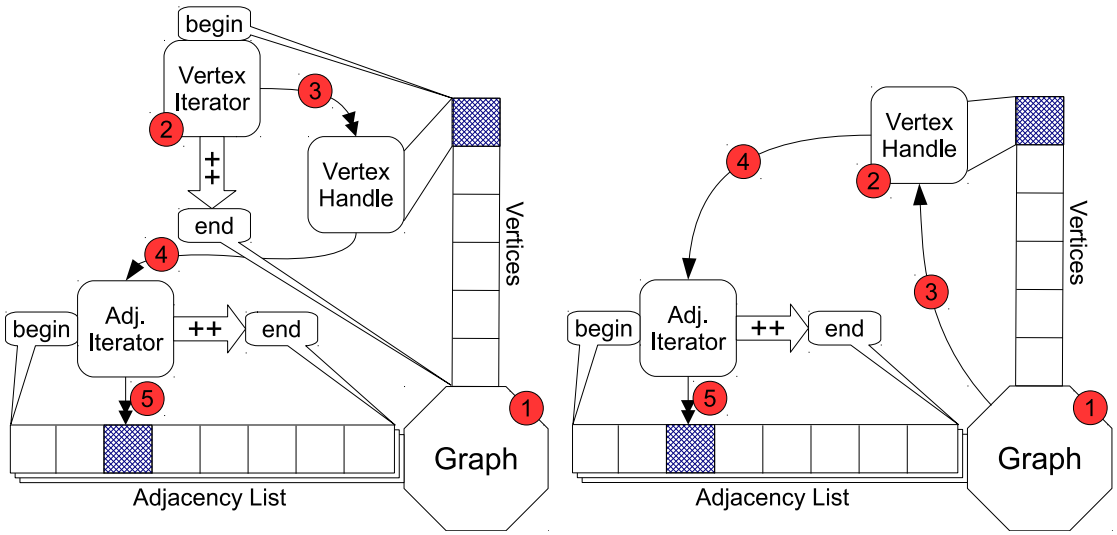
3.2 Abstraction and API

A graph is represented as a `CarafeGraph` object in a directed, adjacency-list-alike format. It can be created by passing a handle to a file containing the raw graph data. The data is imported and placed into distributed memory and a globally accessible address is passed back. Following this, the `CarafeGraph` can be initiated on other machines in the cluster by passing them the address. Each vertex in the graph is assigned an identifier, or *vertex ID*, by the system. IDs are zero-based and contiguous, and there exists a special, invalid vertex ID for use by algorithms, e.g. as sentinel. Vertices contain a *vertex name*, which is specified in the imported file, and a property map, which contains run-time data from algorithms and related contextual information. A vertex is accessed by obtaining a handle object to it, the `VertexHandle`. There are two distinct ways of obtaining `VertexHandle`s, as there are two fundamentally different types of accessing vertices: sequential and random. The access is sequential if it iterates over multiple vertices in the order they are stored internally. If the application imposes another access order, the access is random. To support both as efficiently as possible, the abstraction provides two specialized access modes illustrated in figures 3.1a and 3.1b:

Sequential access From the `CarafeGraph` a begin and an end iterator are obtained. The begin and end can be specified freely by passing two vertex IDs, constrained only by the graph's bounds. Between the iterators is a range of vertices, where the end iterator is after the range and may not be dereferenced. The system assumes that the whole range will be accessed, and will optimize for the range given. On dereferencing the begin iterator, it passes back a reference to a `VertexHandle` belonging to the iterator. The data referenced by the `VertexHandle` can be accessed immediately without any further steps. The next vertex is accessed by incrementing the begin iterator, at which point the `VertexHandle` becomes invalid. When the begin iterator is after the last vertex of the range, it is equal to the end iterator and dereferencing it is invalid.

The API distinguishes between mutable and constant iterator types. Whereas the `VertexHandle` of a mutable iterator is also mutable, i.e. can be written to, the constant iterator and its handle are read-only. The distinction allows the system to optimize read-only accesses.

Random access First, a `VertexHandle` object is constructed. It, together with a vertex ID, can then be passed to the graph object for initialization. Unlike in the sequential access case, the abstraction grants full control over the RStore-data path: the `VertexHandle` provides `map()`, `unmap()`, `read()`, and `write()` functionality. Thus, before vertex data is accessed,



(a) Sequential access: (1) Create a `CarafeGraph`; (2) Get a vertex iterator pair from the `CarafeGraph`; (3) Dereference it to get a `VertexHandle`; (4) Get an adjacency list iterator pair from the `VertexHandle`; (5) Dereference it to get an adjacency

(b) Random access: (1) Create a `CarafeGraph`; (2) Create a `VertexHandle`; (3) Initialize the `VertexHandle` by passing it to `CarafeGraph`, then `map()` and `read()`; (4) Get an adjacency list iterator pair from the `VertexHandle`; (5) Dereference it to get an adjacency

Figure 3.1: Application Workflow in Carafe

the vertex must be mapped and read. If modified, it must be written and after use must be unmapped. After unmapping, the `VertexHandle` can be re-used to access another vertex.

With a `VertexHandle` in hand, the outedges of the vertex can be accessed analogously to sequential vertex access: begin and end iterators are obtained from the `VertexHandle`. Dereferencing the begin iterator logically traverses the edge and gives back the vertex ID of an adjacent vertex. The next outedge is accessed by incrementing the iterator. Outedges are listed in an unspecified order, giving the system freedom to optimize.

Alternative design In an alternative, imperative design, an iterator is initialized similarly to a `VertexHandle`: for a vertex iterator by passing it to the `CarafeGraph` together with a vertex ID, for an edge iterator by passing it to a `VertexHandle`. An edge has an *edge ID* and is represented by an `EdgeHandle`, analogously a `VertexHandle`.

Arguably this could be a better design as it eliminates the vertex-centricity of our chosen one, allowing for edge-centric applications. However, we argue that the object-oriented design is equally powerful, and could be extended with features such as `EdgeHandle`s without compromising the design or impacting performance. To this end, imperative versus object-oriented is a matter of taste.

3.3 Identifying Vertices

To uniquely identify a vertex within the system, every vertex requires an unambiguous identifier. Usually, the file defining the graph also defines vertex IDs for the same purpose in the scope of the file. Henceforth, we will refer to the vertex ID in the system by *vertex ID*, and the vertex ID in the file by *vertex name*. For practical purposes, both the ID and the name are assumed to be non-negative integers.

Besides being unique, a key requirement for the ID is that it should allow the system to find the corresponding vertex with minimal overhead. However, input and output must be meaningful, thus given as vertex names. From the perspective of an algorithm implemented on top of the API, this discussion is irrelevant — only the graph structure matters, which is independent of how a vertex is called, giving us leeway in designing a naming scheme. This poses the question: should the vertex ID and name be identical, or does a better alternative exist?

Let the vertex ID and name be identical in design (1), and in design (2) the system generates an ID for every vertex with a bijective mapping to the vertex name and a translation function must be called explicitly when the vertex name is required. Clearly, design (1) does not need a mapping for input and output. But it does not correlate in any way to the system's internal vertex data structure. Depending on the choice of data structure, it becomes necessary to translate between a vertex's location and its ID. Design (2) is more flexible. For instance, the system has the freedom to choose the naming scheme to suit its internal structure, potentially avoiding a layer of indirection when accessing a vertex. Alternatively, the system could imitate design (1), and the translation function would be the identity function. We argue that vertex lookup during computation is much more frequent than input and output, thus translating input and output is a small price to pay for the additional flexibility given to the system in design (2).

There is a second choice to make: should the vertex IDs be contiguous or non-contiguous? A contiguous naming scheme is an additional guarantee and therefore less flexible. For instance, if the graph structure is mutable, removing a vertex would be difficult to handle as it would leave a "hole" in the list of vertex IDs. Since the primary focus of this work is performance, we constrain Carafe to graphs with immutable structure. On the positive side, contiguity is practical for performing arithmetic on vertex IDs, e.g. when setting a range of vertices for the begin and end iterators, the end's vertex ID can be found by a simple addition. For our system with the stated constraints, we have chosen to sacrifice some of design (2)'s flexibility for the convenience of contiguous vertex IDs.

3.4 Graph Layout and Storage

High-level graph data structure As explained in section 2.4, choosing a graph data structure comes with many trade-offs. When targeting large graphs, however, storage size quickly becomes the predominant consideration. Given millions or even billions of vertices and edges and the goal of storing sparse, power-law graphs, non-linear space complexity is infeasible, and the constant factor plays a significant role as well. Although now only a consideration of second priority, time complexity for operations on the graph is also important. As not all operations are equally important, the most frequently used can be prioritized. In the case of Carafe, these are finding a vertex, and vertex and neighbor iteration.

There are only two fundamental data structures with linear space complexity, the adjacency list and the edge list. Although the latter appears to have slightly better properties as it is not dependent on the number of vertices, the alleged advantage comes from not being able to store vertices without at least one edge, which is actually a disadvantage. In practice the former has a lower constant factor, being almost twice as compact. The adjacency list's second advantage is in its optimal time complexity for the stated key operations, and in its adaptability to partially trade these off for bolstering other operations, if desired.

Considering all of the above trade-offs, we choose the adjacency list to be Carafe's underlying data structure.

Storage medium An adjacency list is only a method for organizing the components of a graph; beneath it must be a supportive data structure to store the components. This supportive data structure ultimately depends on the storage medium. As per the design goals, the graph must be stored distributedly and globally accessible, and the assumption of fast remote access must

be fulfilled. In a design (1), Carafe comes with its own distributed storage solution. However, designing a distributed storage framework is by itself a non-trivial undertaking, calling for an evaluation of existing frameworks. A design (2) builds on an existing distributed key-value store. Vertex IDs serve as keys, the value being the vertex's adjacency list, property map, and other data associated with the vertex. A last design, design (3), uses an existing distributed in-memory storage framework as represented by RStore (see section 2.2), which emulates a byte-addressable array and makes no assumptions on the data layout. While design (2) is convenient in that it is content-addressable, current key-value stores violate the data and control path principle[3]. A further drawback is that key-value stores offer little or no control over data locality, limiting our freedom in design and implementation. In design (3), RStore performs no data caching, only RDMA resource caching. As its distributed memory is byte-addressable it gives full control (and full burden) of data layout, and hence locality, to Carafe.

We choose design (3) because it provides us with the desired flexibility in design and implementation, but at the same time spares us the complexities of implementing our own distributed storage solution.

Low-level supportive data structure Given a byte-addressable memory abstraction, the adjacency list requires an underlying data structure capable of supporting the key operations of vertex lookup, and vertex and neighbor iteration efficiently on a remotely stored graph. As these operations include both sequential and random accesses, the structure must be efficiently tolerant of both patterns while holding only a small part of the graph in local memory. And given the goal of large graph storage, the structure must retain the spacial efficiency of the adjacency list with only minimal overhead.

The most basic design (1) is based on arrays. Because every vertex's adjacency list potentially has a different length, concatenating the vertex's adjacency list to the vertex makes the vertex un-indexable, effectively degenerating it to an unordered list. Therefore, the vertices must be stored separately from their adjacency lists, where each vertex is located at the position given by its ID. The adjacency lists are stored in a separate array; each vertex holds a reference to its start together with its size. Inside an adjacency list, the individual adjacencies are unordered. A second design (2) stores vertices in a B+tree and stems from B+trees' use in file and database systems. The adjacency lists are stored as in design (1). Linked data structures, such as linked-lists or binary trees, and hash maps are not considered due to their poor locality for accessing a range of given vertex IDs, requiring excessive remote accesses. We do not consider resizing operations. A third design (3) builds on these ideas, but segregates the arrays into multiple blocks of fixed size. Effectively, design (3) is a 2-level hierarchy of arrays.

Design (1) works well if vertex IDs are contiguous. Random accesses are efficient, as they only require a single access to the structure. Sequential accesses are also efficient, as data can be pre-fetched; any arbitrary range of given vertex IDs is stored contiguously, as is the adjacency list. Accessing the adjacency list of a vertex requires two accesses, one to the vertex and one to the adjacency list. In design (2), vertex IDs can be arbitrary, although lookups within a node effectively become indexed lookups if IDs are contiguous. A B+tree is advantageous over a B-tree as internal nodes contain only indices, allowing for higher order nodes to make the tree flatter, and sequential accesses can be accelerated by linking and pre-fetching leaf nodes. Compared to design (1), the B+tree allows for tuning the node size to a multiple of RStore's internal block size, which avoids external fragmentation. The fixed-size nodes can be cached, contrary to arbitrary segments of array data, caching of which leads to an infeasible amount of metadata. Keeping nodes close to the root and the root node itself in cache is feasible, making many random accesses possible in one or two remote accesses. However, a B+tree is considerably more complicated to implement than a flat array, requires more remote accesses, and degrades the $O(1)$ vertex lookup complexity of the flat adjacency list data structure to $O(log(n))$. A compromise between these first two designs is design (3). The array structure

requires contiguous IDs, and is indexable without intermediate lookups — the block and offset are calculated by integer division. Sequential accesses have locality as in arrays. Fixed-size blocks are cacheable and pre-fetchable as with a B+tree. We argue that for an immutable, remote graph structure, design (3) is an optimal solution.

Vertex Property Maps Alongside the graph structure, per vertex property maps must be stored. Each of them is a logical part of the vertex, and is mutable in the sense that the map's fields can be modified. There are two ways to store them: either they are embedded in the graph structure from above or they are stored in a separate data structure alongside the graph, referred to as *internal* and *external property maps*.

Internal property maps have a locality advantage compared to external maps; when a vertex is loaded from remote storage, the same access can also load the property map. They are also simple to implement, being another field in the vertex. On the flipside, they have three significant drawbacks. First, embedding property maps into the graph structure fixes the map's size, making the design inflexible to accommodate different contexts and requiring a re-import of the whole graph to switch contexts. Second, the embedded map design does not allow for multiple contexts to exist simultaneously; only one can be embedded in the graph structure at any one time. Third, an algorithm may not access the property map on each vertex access, instead only traversing the graph structure. In that case, binding the property map to the graph is inefficient, leading either to non-contiguous accesses or to superfluous data being read or written.

External property maps are free of these three drawbacks. This comes at the cost of managing the property maps in their own data structure, e.g. stored in the same way as vertices. Touching a property thus requires at least one more remote access, depending on the data structure design chosen.

We argue that in the short term, internal property maps are sufficient. But in the long term, when this project reaches maturity, their drawbacks outweigh their advantages, and an alternate design must be considered.

3.5 Caching and Pre-Fetching Strategy

In a scenario where data is held in a remote location, access to said data is by definition slower than to data held locally. A common method to reduce the number of such remote accesses is to predict how data will be accessed in future, made possible by spatially and temporally local access patterns. If the data is being read, a previous read to the same data can be retained locally — this is *read caching* — or *pre-fetched* ahead of time. If the data is being written, multiple writes can be cached and then *written back* in one go — this is *write caching*.

In offline graph processing models, such as the Pregel model, the dataset is divided into multiple partitions. The partitions are split up among multiple machines for processing, where each machine "owns" one or more partitions. The owner has exclusive access to its partitions. This allows for a better approach to caching. If the owned partitions fit into local memory, the whole partition can be pre-fetched once and subsequently read-cached. Write caching is possible, too, but if the model does not require synchronization, there may not be a pre-specified time when the write-back must happen. In this case the system must choose a suitable time for writing back.

Online graph processing models are, in their pure form, more difficult to cache than offline models, as the underlying data may change while the computation is ongoing. However, with our constraints the problem is reduced to predicting the access pattern of the algorithm in question. Dijkstra's algorithm, for instance, has spacial locality when iterating the neighbor list of a vertex. It has temporal locality as well, when it places neighboring vertices into the active

set and later removes them again. However, in the worst case nearly all vertices are in the active set, caching all of them is potentially infeasible in large graphs.

In the infinitely large design space of caching, there is a lower and an upper bound. The lower bound is to cache nothing, which means to either rely on fast enough access to storage or to cache in a lower or higher layer. The upper bound is to greedily cache as much as fits into local memory — let this be *maximal caching*. On the one hand, the lower bound does not absorb any accesses. On the other hand, the upper bound may occupy memory better utilized in the application or a different part of the system. A way to reduce the latter problem is to bound the cache's size to a given amount of memory. This poses the question of how to define the amount of memory given to the cache. While a definite answer to that question is elusive, just above the extreme lower bound is an interesting answer calling for closer consideration: the cache is bounded to a size just large enough to do pre-fetching, write-back, and immediate temporal caching (i.e. when the cached values are used several times in rapid succession) — let this be *minimal caching*.

Accepting that there are a wide variety of cache types with very different requirements in their combinations and applications, combined with our aim of supporting online and offline graph processing models, we argue that designing for read caching with pre-fetching and write-caching with write-back is necessary. As finding an optimal cache size for all applications and inputs is not possible in the general case, we look into the minimal and maximal strategies in our evaluation (refer to section 7.2). From the results we draw our conclusions.

3.6 Fault Tolerance

Failure of an application built on the Carafe API can only affect the context stored in the vertex property maps of the affected application. The graph structure remains intact at all times for other applications, as it is immutable and read-only access is sufficient. When an application fails during the process of writing within the property map, two cases must be differentiated due to the caching design of Carafe: (a) If the application is using write caching and no write-back is ongoing at the time of failure, the data in cache is lost and the remotely stored data remains consistent. (b) Otherwise, if a write-back is ongoing or the application is not using write caching, thus remote data is being modified, the state of the data under access will be inconsistent due to the zero-copy, in-place modification nature of Carafe, and RStore beneath it. This scenario of inconsistent state can be avoided by enabling the optional copy-on-write mode provided by RStore, which modifies data in a new location and transparently updates the logical address when and if the write completes successfully. In case of failure, the original data is left intact.

Beyond data consistency, Carafe does not offer support in failure recovery as the primary goal is to provide a graph data abstraction. Applications must include their own, context-dependent means of recovering from a failure.

4 Carafe Implementation

Carafe is implemented as a library in $\sim$ 2500 lines[1] of templated C++ code. A description of the implementation follows.

4.1 Graph Storage Format

The Graph A graph is stored remotely in RStore in a serialized form, is persistent, globally visible and accessible. A `CarafeGraph` object stores contextual information about the graph and metadata necessary to bootstrap the graph when loaded. Metadata includes the graph's name in string format, a serial number specifying the type of the embedded property map, and the RStore addresses and sizes of the vertex block array and adjacency list block array. The `CarafeGraph` object is stored in an RStore namespace named *graph namespace*. The RStore address of the `CarafeGraph` object together with the namespace is the only knowledge necessary to load the graph.

Vertices are stored as `SerializedVertex` objects in one block array in the *vertex namespace*, with their vertex ID being the position index. Each `SerializedVertex` stores its own ID for back-referencing, its own vertex name for reverse translating ID to name, the offset and size of its adjacency list, and its embedded property map. The vertex ID and vertex name are 64-bit, unsigned integers instead of a more space-efficient 32-bit format to support graphs with more than 4.3 billion ($\approx 2^{32} - 1$) vertices. The property map is a static type passed to the vertex as a template parameter and is required to be in a serialized form of the application's choice.

Edges are stored as the vertex IDs of their destinations in an adjacency list per vertex in the *outedge namespace*. All adjacency lists in the graph are stored concatenated to each other in one block array, i.e. the block array is logically seen as one array of 64-bit, unsigned integers.

Serialization Serialized data is required to be plain old data (POD)[2], as defined by the C++11 standard[13], have no local pointers, and little-endian byte order. If the data is a struct or class, it must be packed to avoid compiler- and operating system-specific padding[14, p.198ff]. The aim of these measures is to ensure compiler and language interoperability with minimal run-time performance overhead.

Block Array The block array data structure contains blocks of fixed size. The size is set to a multiple of the RStore chunk size and blocks are aligned to RStore chunks. Each block is given a *block ID*, which is calculated by integer-dividing the vertex ID by the block size — the vertex IDs are required to be contiguous and zero-based. For each block an `raddress` object is stored in a metadata array at the position of its block ID. When the block is mapped, the block ID is calculated from the given vertex ID. The vertex's offset within the block is obtained by calculating the modulo of vertex ID and block size. Lastly, the `raddress` is looked up and mapped. The other operations are performed analogously, although while blocks are always mapped as a whole, reads and writes are by unit of the type stored in the block array (e.g. `uint64_t`). Storing `raddresses` by block instead of by unit stored within the block saves

[1] As counted by CLOC v.1.53 (`http://cloc.sourceforge.net`)

[2] In essence, POD must be a scalar type or a collection thereof, which excludes reference types and language features such as virtual functions and virtual base classes.

memory used for metadata. Newer versions of RStore support sub-`raddress` mapping, which would make using a single `raddress` per block array possible.

Block Caching Blocks are cached for (a) mapping, (b) reading, and (c) writing. If a block is frequently accessed, map caching keeps it in memory, unmodified from the last access, as opposed to unmapping and then remapping it in RStore. Although RStore does resource caching, which includes memory chunks, it takes time to locate the cached resource and set up metadata.

Closely related to map caching is read caching, as read caching implies map caching. When any location within a block is read, the whole block is read from its remote location — this is pre-fetching. Subsequent accesses within the same block are read from cache.

Write caching is similar; writes to a block are not written back until the block is unmapped. This has a catch: an RStore map does not automatically perform a read. Thus a mapped, unread block holds undefined data. If only a part of the block is accessed and then the whole block is written back, valid data in the remote block location will be overwritten by the undefined data in the local block. Therefore, the system reads the block following a map before allowing any write operations.

Hypothetically, the system could remember which segments within a block were written to — are *dirty* — and only write back the dirty segments on unmapping the block. In practice, this is infeasible as the amount of metadata needed for remembering dirty segments easily exceeds the size of the block; imagine an a-b-a-b scenario in which an integer in every vertex within a range is modified, while the remainder of the vertex is unchanged; the beginning and end of every individual, dirty integer would have to be remembered.

In case a write access must immediately be written back to remote storage, the system's API offers a method to force an arbitrarily-sized write of byte-granularity through the write cache.

The force-write's counterpart is a force-read, which reads an arbitrary, mapped segment from remote storage, ignoring the read cache. Of all these caches, only map caching is compulsory. Read caching is enabled by default as data is expected to be mostly static and only read by the mutator. Write caching is disabled by default as it causes conflicts in case of multiple mutators modifying the same block in disjunct segments.

Block Eviction Cached blocks are evicted as per the clock algorithm[15]: blocks are stored in a circular list of slots. The references of each block are counted in a reference counter r. And each block is given multiple *chances* before being evicted; the chances remaining are counted in a chance counter c. A pointer, called the *clock hand*, points to one of the cache's slots.

When `map()` is called on a block not currently in cache, the slot pointed to by the the clock hand is examined. If the reference counter $r \neq 0$, the clock hand is moved forward by one slot and the next slot is examined. If the block is not currently referenced ($r = 0$), but still has at least one remaining chance ($c > 0$), then c is decremented by 1 before the clock hand is moved forward. Else, the block in the slot pointed to by the clock hand is chosen for eviction.

Once a victim is found, the new block is installed in the chosen slot. The reference counter is set to 1 and the chance counter is set to a constant k; in our implementation, $k = 1$.

If `map()` is called on a currently not referenced block already in cache, its chance counter is restored to k. In case all cached blocks are referenced and none is evictable, the system throws an exception and escapes.

The clock algorithm is an approximation to a *least recently used (LRU)* eviction policy, as LRU is notoriously difficult to implement efficiently. The rotating pointer resembles the hand of a clock, hence the algorithm's name.

4.2 Graph Access

With a `CarafeGraph` object in hand, the vertices of a serialized graph in RStore are accessed by random or sequential means:

Random Vertex Access The `VertexHandle` effectively implements a "Bridge" pattern[16, p. 151ff] to provide a clean, abstracted API to the underlying `SerializedVertex`. When the `VertexHandle` is passed to `CarafeGraph` with a vertex ID, the vertex ID is placed into the `VertexHandle` and a local pointer to the containing block array is set.

When `map()` is called on the handle, it forwards the call to the block array with the vertex ID as argument. The block array calculates the block ID and offset of the vertex, maps the vertex's block if it is not in cache, increments its reference counter, and returns a local pointer to the block and the offset. The offset is given in units of `SerializedVertex`.

On a `read()` call, it is passed through to the block array with the vertex ID and block pointer. The vertex ID is again used to calculate the block ID. The cache is queried if the block has already been read. If so, the call returns immediately, if not the block array is instructed to read the block before returning. `write()` is analog to `read()`, the only difference being that a cached write marks the block as dirty.

`unmap()` drops down to the block array with the vertex ID, finds the block, and decrements the reference counter. When the reference counter reaches zero, the block remains mapped and in the cache, but is evictable. Finally, the block pointer and offset variables are invalidated.

Sequential Vertex Iteration When a `HandleIterator` is obtained from the `CarafeGraph`, it is constructed with a `ManagedVertexHandle` and a `BlockIterator`. The `BlockIterator` is constructed with a pointer to the vertex block array, a vertex ID, and the size of the range to be iterated as parameters. In the case of vertices, the vertex ID is equivalent to the position within the array.

On being dereferenced, the iterator dereferences the `BlockIterator`, which checks if it already has a pointer to the current vertex's containing block. If yes, it returns a reference to the current vertex's `SerializedVertex`. Otherwise, it first maps and reads the block. The read uses the range's size to intelligently determine if the whole block is within the range or not; if not, only the block segment which is within the range is read (although if read caching is enabled, the whole block is read anyway). The range is read with one `read()` call, which takes the vertex ID, block pointer, and the size of the range. The `HandleIterator` passes the reference to the `ManagedVertexHandle` and returns a reference to the latter.

On being incremented, the iterator hands the call down to the `BlockIterator`. This one increments the block offset and checks if the offset is valid. If it is, it then increments the vertex ID and returns. If not, it writes the block, analogously to the read described above including the size optimization, and unmaps the block. Constant iterators, which are read-only, omit the write and only unmap. Only then does it increment the vertex ID and return. The write-back and unmap of the last block in the range happens when the iterator is destructed.

Iterator comparisons on the `HandleIterator` are passed down to the `BlockIterator`, and are implemented as a comparison on the current vertex IDs of the current and other iterator.

The `ManagedVertexHandle` is equivalent to the random-access `VertexHandle`, but does not include the `map()`, `unmap()`, `read()`, and `write()` functions.

Sequential Adjacency List Iteration Like with sequential vertex iteration, the adjacency list of outedges are iterated with a `BlockIterator`, which is however not wrapped.

The iterator is obtained from the vertex handle, which can be either a `VertexHandle` or a `ManagedVertexHandle`. On construction, a pointer to the adjacency list block array, the start index, and adjacency list's size are passed as parameters.

When dereferenced, the iterator returns a vertex ID reference. The remaining operations are as explained in sequential vertex iteration.

4.3 Importing a Graph

Edge List Format The most common format graphs are distributed in is an edge list; each edge is represented by the two vertices at either end, with vertices encoded as ASCII numbers separated by whitespace. Edges are assumed to be directed.

Efficiently importing this edge list format into Carafe is challenging; edges can be listed in any order, the number of vertices and edges in the graph is initially unknown, as is the number of adjacencies per vertex. Thus memory for the graph cannot be pre-allocated immediately, nor can adjacency list positions be calculated within the adjacency list block array. Also, reading in an edge is effectively a random access within the graph, although in some files edges are sorted by the edge's head endpoint in ascending order (all outedges of a vertex are grouped), which makes the access sequential.

The importer must therefore make two passes; the first pass calculates the graph's in-memory size and component placements, the second pass imports the data into Carafe. In the first pass, a hash map mapping vertex names to vertex ID and out degree is populated, and the number of vertices and edges is counted. Vertex IDs are generated here, where vertices are ID-ed in the order they are encountered (i.e. the i'th vertex gets ID i), regardless if they are the head or tail endpoint of an edge. In the second pass, space in RStore is reserved and allocated. Then the graph is iterated sequentially with a `HandleIterator`, and the vertices are populated from the hash map. Finally, the file is read again to populate the edges. Here, the head endpoint must be mapped and read to find the current end position in the adjacency list of the vertex and to increment the list's size. The vertex is kept mapped until a different head endpoint is encountered. During both passes, the file is read sequentially with no random accesses.

The import process can be sped up considerably by enabling read and write caching — the `HandleIterator` writes back vertices block-wise either way, but the write back of edges is faster due to less re-maps of vertices and block-wise adjacency list reads and writes.

Fast Format Despite the performance enhancements for the edge list file format described above, there is much potential for improving the import time due to the number of remote operations on RStore. We reason that a graph will be imported into Carafe multiple times, with a conversion from edge list format to a custom format taking place only once. Therefore Carafe has a custom file format, dubbed "fast format", with the aim of achieving optimal import performance.

A fast format file contains an array of 64-bit, unsigned integers in little-endian byte order. The graph is stored in an adjacency list data structure as follows:

Metadata At position zero is the format version number, at positions one and two are the number of vertices and edges, respectively.

Vertices From position + *metadata* are the vertices, in order of ascending IDs. Each vertex stores its name, outdegree, and indegree in the stated order.

Edges From position + *vertices* are the edges, grouped by vertex with the groups ordered equally to the vertices. The start offset for each adjacency list is the cumulative sum of the adjacency lists' sizes (= outdegrees) before it.

The converter goes through the same steps as the edge list importer, but stores the graph to an `mmap()`'ed output file in fast format.

The importer is more simple. It first `mmap()`'s the file. Then it reads the metadata and reserves and allocates the graph in RStore. Following that, it populates the vertices with a

`HandleIterator`. Finally, it populates the raw adjacency list block array with a `BlockIterator`. Read and write caching have no effect on this importer, as the iterator operates block-wise even without them.

The performance of the fast format importer is much better than that of the edge list importer. Refer to section 7.3 for an evaluation of measurement results.

5 C-Pregel Design

5.1 Design Goals

C-Pregel is conceived as a high-performance implementation of a Pregel-alike API[1] built on our Carafe graph processing framework. It is a showcase for the generality of Carafe's interface, demonstrating how a high-level, distributed graph processing framework is constructed with the primitive graph operations exposed by Carafe.

The Pregel model is an embarrassingly parallel graph processing model (see section 2.6.1). This provides frameworks implementing the model the potential for large-scale distribution of computations. The first challenge in this comes from attaining high throughput to dynamically fetch graph components from remote memory when they are not held in local memory (e.g. due to the graph's size), and exchanging messages between cluster nodes before synchronizing the superstep transition. In large graphs there is enough parallel slack to hide latency while processing vertices, if the pipeline delivering unprocessed vertices remains fed. Attaining high throughput is thus key to high performance.

The second challenge originates from the distributed setting of C-Pregel. Minimizing the runtime applies not to a single node locally, but globally to the whole cluster; nodes must work together by distributing the workload evenly across the cluster. With a load-balancing solution we must ensure that individual hardware components, such as CPU cores, network links, or DRAM, do not become a performance and capacity bottleneck for the entire system.

Our design goals for C-Pregel are:

Semantic Pregel Compatibility The framework should offer all necessary functionality to run applications assuming the Pregel model. Applications written against other frameworks should semantically be portable to C-Pregel.

High Throughput The data and control path separation and zero-copy design of Carafe should be extended to C-Pregel. The system should achieve high vertex and message throughput by leveraging these properties into a distributed processing environment. It should score optimal performance in the Carafe setting of high-performance network hardware and multi-machine, multi-core, rack-scale cluster landscape.

Zero Graph Partitioning Instead of relying on an NP-hard graph partitioning problem to achieve good performance in selected graphs and algorithms, the system should focus on achieving excellent performance in all, especially also in difficult, scenarios. By not attempting to minimize intra-cluster communications via graph partitioning, balancing of the cluster nodes with polynomial-time workload partitioning attains C-Pregel's full attention.

The scope of our design is limited to the core Pregel graph processing model; supplementary features such as message combiners and global aggregate functions are not supported. Furthermore, C-Pregel is limited by the feature set of Carafe; in particular, structural graph mutations are not possible. Due to the design constraints inherited from RStore's resource pre-allocation and our zero-copy design goal, arbitrary message destinations are not possible; a vertex can send messages only to its neighbors, as defined by the graph's structure.

[1] Google's Pregel API is not public; the API in the paper is merely an abstraction.

5.2 Abstraction and API

On a high level, the C-Pregel closely resembles that of Google's Pregel implementation. There are, however, a number of differentiating factors which must be accounted for.

As in Pregel, the `Vertex` class is the API's core component, and is subclassed by the application. `Vertex` is a template class and does not specify a `compute()` method; the application must provide it itself. The application must pass the concrete vertex type it defines as a template parameter to the system, which instantiates the concrete vertex type at a time of its choosing. No guarantees are given concerning the maintenance of state of any application-defined fields of `Vertex`. This template-based design is a matter of preference; any effects on performance when compared to a dynamic-dispatch-based design are a secondary consideration.

Different from Pregel is the vertex initialization. In Pregel, the system sends an initial activation message to each vertex. In Carafe, the system calls an `initialize()` method to initialize the vertex property map. In the same fashion as with `compute()`, `initialize()` must be defined by the concrete vertex type. `initialize()` is guaranteed to be called exactly once on every vertex in the graph. While the Pregel approach is more elegant from a system design perspective, it forces `compute()` to handle the first superstep as a special case. In this sense, our approach is more practically oriented.

5.3 Distributed Architecture

C-Pregel is a distributed framework. As such, its nodes must coordinate their initialization, workload distribution, supersteps, and other information among another. There are two fundamental design choices for accomplishing this; let these be designs (1) and (2).

(1) is a master–worker design. A unique master has a global view of the system and handles all coordination within it. Multiple workers are conducted exclusively by the master and never communicate directly with another.

(2) is a fully distributed design. All nodes are equally responsible for coordination within the system. They must be mutually consent for global action, such as beginning a new superstep.

Although the master in design (1) constitutes an inherent single point of failure, whether design (2) possesses more efficient failure recovery is debatable. From a design perspective, design (1) is more simple than design (2). Simplicity in design clearly outweighs alleged robustness in failure recovery. Furthermore, in the scope of our work, the scale of the cluster leads us to assume a low probability of failure. In our decision to favor simplicity we are supported by other systems[17, 10] and recent research in consensus protocols[18].

5.4 Processing Vertices

Given a master–worker architecture, the steps for processing vertices are as follows:

1. Register the workers with the master and distribute work to them in the form of vertices. Synchronize all workers.

2. Initialize all vertices by calling the application-provided `initialize()` method on each vertex. Activate all vertices.

3. Fetch the incoming messages for the current superstep. Activate all inactive message destinations.

4. Call the application-provided `compute()` on each active vertex.

5. Post the outgoing messages of the current superstep.

6. Synchronize all workers. If there are any active vertices or outgoing messages, transition to the next superstep by repeating from step 3.

The steps of special interest are the registration and synchronization of workers, the work distribution, and the message fetch and post. Registration and synchronization are explained in the next section. Work distribution is explained in section 5.6. The workings of message passing is detailed in section 5.7.

5.5 Superstep Synchronization

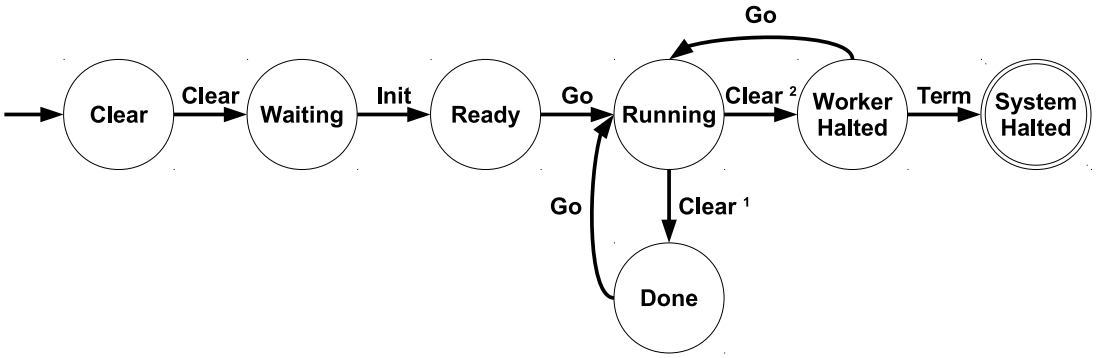


Figure 5.1: State Machine of a Worker for Superstep Synchronization;
The `clear` state transition from `running` depends on the worker having active vertices or sent messages (`clear`^{1}), or neither (`clear`^{2}).

For the system to implement the Pregel model correctly, workers must be synchronized before the first superstep and after every superstep thereafter. To this purpose, the master and workers implement a state machine. For the state machine to be correct, a set of rules must be obeyed. Firstly, the workers are in exactly one state at any time. They transition atomically from one state to another only when the master issues a directive. Directives are issued atomically and only one directive can be outstanding at any one time; all workers must have transitioned to the new state before the next directive can be issued. Lastly, a directive issued must be legal for the current state; a directive is legal if it defines a transition from the current state to another state.

In the state machine, the master ensures that both the directive and the workers' state are `clear` on initialization. Only then may the workers be started; they immediately set their state to `waiting`. In state `waiting` the master communicates the graph's location, partitions and distributes the workload, initializes a messaging framework, and directs `initialize`. The workers now load the graph, read their workload partition, hook into the messaging framework, and transition to state `ready`. This state signals that they are ready to start a superstep.

The master directs `go` to start the first superstep. On starting a superstep, the workers immediately transition to state `running`. Once all workers are `running`, the master sets the directive to `clear`. This intermediate state ensures that the workers do not mistakenly read the same `go` directive twice. When the workers have completed their work for the current superstep, each worker receives the `clear` directive and then transitions to one of two states: (a) `done` if it has active vertices or outgoing messages in its workload partition, (b) `halted` if all vertices in its workload partition are inactive and it has no outgoing messages. If all workers are in state `halted`, the master directs `terminate`. Else, the master starts the next superstep by directing `go` once again.

5.6 Workload Partitioning

The goal of the workload partitioning is to use all resources available to the system such that application runtime is reduced to a minimum. We differentiate workload partitioning from the NP-hard graph partitioning problem by its goals. Like graph partitioning, the goal of workload partitioning is to balance the workload across all partitions such that utilization of available system resources is maximized. Unlike graph partitioning, workload partitioning does not aim to minimize inter-partition communication.

To bound implementation complexity, we introduce an additional design constraint: workload can only be distributed once during startup of the system. Effectively, this is a static scheduling problem for which we have developed several candidate solutions:

By Vertices Distribution The most trivial method of distributing the workload is by splitting it into equally sized partitions by vertices. If all n partitions have the same size and there remain some number $m < n$ vertices left over, these m vertices are distributed among m partitions.

By Edge Distribution Somewhat similar to by vertex distribution, by edge distribution attempts to split the workload evenly into partitions by the number of edges. However, as the Pregel model is vertex-oriented, edges belonging to the same vertex must be assigned to the same partition; in other words, the distribution is by vertex, but vertices are weighted by the number of their edges. The weighting can be either by outedges or by inedges.

The algorithm is simple and greedy: Find the ideal partition size S_{ideal} by dividing the number of edges by the number of partitions. Iterate through the graph's vertices and sum the number of edges encountered. When the sum exceeds S_{ideal}, reset the sum, group the vertices encountered since the previous sum reset to a partition, and continue iterating.

By Component Weight Distribution The edge distribution algorithm can be taken one step further. Instead of only counting edges, the weight factor is made explicit and the two types of graph components are considered: vertices and edges. Each of these is assigned a given weight. Then the weights are applied to the algorithm above.

Both the vertex and edge distributions are quickly skewed by imbalances in the graph. Let there be a graph in which few vertices have many edges and many vertices have few edges, and the graph is stored with vertices sorted by their adjacency list size in ascending order. Then the first partition in by vertex distribution will have only few edges, and the last partition will have many edges. Given an algorithm that iterates the adjacency list of every vertex, the runtime of the last partition will be significantly longer than that of the first partition. Thus the workload is imbalanced. The imbalance is inverted if the workload is distributed by edges.

The skew is avoided by the by component weight distribution. Both vertices and edges are considered. Their weights represent the cost of the system to process them. For vertices, the cost factors in the computational resources used to fetch the vertex, evaluate if it is active, execute the `compute()` method, et cetera. For edges, it is the resources used to inspect the edge during an adjacency list iteration, post and fetch a message traversing the edge, process the message, and so forth. Consequently, the vertex and edge weights can be adjusted to set these costs in proportion to one another.

We evaluate the different workload partitioning methods described above in section 7.2, and conclude which is the best choice for highly-skewed, power-law graphs.

5.7 Message Passing

Message passing between vertices is the central component of the Pregel model. It is the reason why supersteps must be synchronized. Most importantly, message passing is the bottleneck of the model. If the graph can not be partitioned such that the number of edges spanning partitions is small, the network traffic compulsorily increases. This stands in stark contrast to network traffic caused by cache misses on the graph structure and associated contextual data, which in the Pregel model can be reduced or eliminated by increasing the cache size. Thus for the system to perform well, it is crucial to have a high-performance message passing subsystem.

To support arbitrary algorithms in the Pregel model, the message passing system is required to (a) have the ability to deliver messages individually, and (b) be delivered in the superstep imminently succeeding the one they are sent in. The first requirement is not necessary for all algorithms, but any relaxations are case optimizations; the system must still support the base case. Given these requirements, our stated goal is to support the base case efficiently. There are a number of different design considerations taken into account.

Design 1 Messages can be passed along edges in the direction of the edge, i.e. a messages traverses an edge. Given that a message must be buffered between being sent and being received, message passing is closely related to the storage of edge property maps. Therefore in a design (1), message passing could re-use the fundamental graph data structures introduced in section 2.4; after all, they are looking up an edge to store a message, and later `compute()` iterates a vertex's inedges to fetch them.

In the current line of thought, there is one important, implicit assumption: every edge is traversed by a message in every superstep. This assumption does not hold, as not all algorithms perform all-to-all messaging in every superstep (although there are some which do, for instance PageRank in section 2.5.2). The implication is that iterating all inedges to check whether or not the edge holds a message in the current superstep is inefficient.

Design 2 A second, more efficient design (2) extends the message passing model to the workers. Let messages between vertices be known as $v - messages$, and messages between workers as $n - messages$. When the vertex u on the worker A sends a v-message to the vertex v on the worker B, the v-message is encapsulated in a n-message from A to B. To reduce the number of n-messages between A and B, multiple v-messages are bundled into one n-message. On receiving a n-message, worker B unbundles the v-message by copying it to a buffer containing messages for vertex v. Thus all messages for v can be placed contiguously into the buffer. When `compute()` iterates the buffer, only v-messages that are present and valid are read; there are no empty slots in between valid v-messages as in design (1).

Unfortunately, there is a data copy in this design, requiring an additional memory and contradicting the stated zero-copy design goal. Also, if implemented as a producer-consumer queue in RStore or with RDMA directly, there must be a read-write barrier between the producer and the consumer. The producer must wait if its production rate is faster than the consumer's consumption rate, and vice versa. The issue becomes especially pronounced when the queue size $q \ll |E_p|$.

Alternatively, the iteration searches all n-messages for v-messages belonging to the current vertex. The copy and additional memory are avoided at the expense of $O(|V_p| \cdot |E_p|)$ time complexity, with V_p and E_p being the vertex and edge sets of worker B's workload partition.

Design 3 A third design (3) bundles all v-messages going from vertices on worker A to vertices on worker B into one single n-message, called a *message container*. The v-messages within a container are then grouped by destination vertex, such that all v-messages within the same

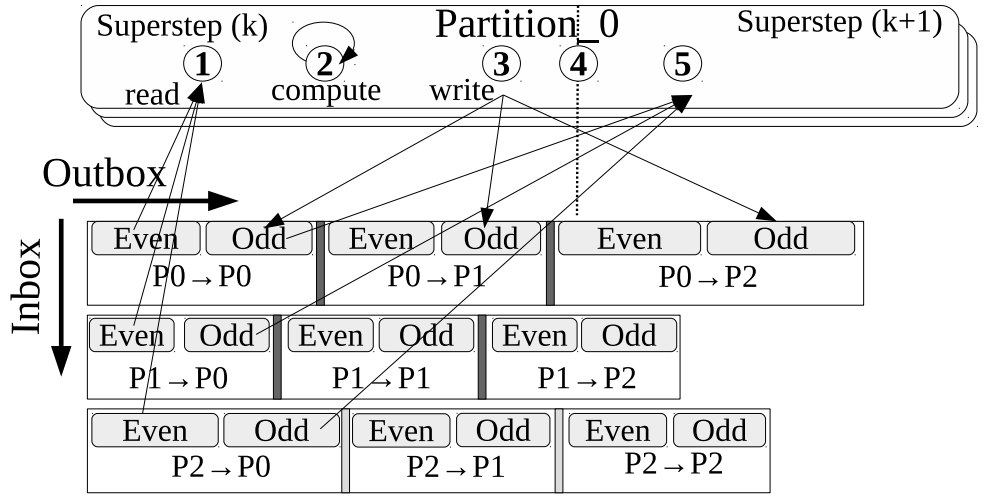


Figure 5.2: Shared Mailbox Model in C-Pregel: Let superstep k be odd; then worker 0 (1) reads incoming messages from even areas; (2) computes the partition's vertices; (3) writes outgoing messages to odd areas; (4) synchronizes between supersteps; (5) proceeds to the even superstep $k + 1$.

container belonging to a vertex v are contiguous. In addition, the groups are sorted such that they are in the same order as the vertices `compute()` is called on.

The first vertex to be processed, u_1, sequentially iterates its v-messages in each container on worker B. u_1 leaves the iterators for each container at the first position after its groups of messages. The next vertex u_2 then re-uses those same iterators to sequentially iterate its groups of messages, and so forth.

The analysis for design (3) is more involved than for the first two. The grouping and sorting can be summarized to one sorting operation per container, where v-messages are sorted by destination vertex ID. Sorting messages can be done in-situ (i.e. requiring no additional memory allocations), in

$$O(|E_{c_1}|\log(|E_{c_1}|) + |E_{c_2}|\log(|E_{c_2}|) + \ldots + |E_{c_n}|\log(|E_{c_n}|))$$

time, with E_{c_i} being the set of edges (= messages) in container i, E_p the set of edges in the worker's workload partition, and

$$\bigcup_{1}^{n} E_{c_i} \equiv E_p$$

assuming an $O(n\log(n))$ sorting algorithm. The total time to iterate the messages is in $O(|E_p|)$. Regrettably, sorting messages precludes a zero-copy implementation.

We argue that design (3) is a good trade-off between memory usage and time complexity, although it is one of the more difficult designs to implement.

Shared Mailbox Model Having settled on design (3) for delivering messages between workers, there remain some details to be worked out concerning how message delivery is organized system-wide.

An edge between the vertices u and v can span any possible pair of workers, because it is not defined where u and v are located. Thus all combinations must be supported by the system. Assuming there are N workers, this implies that there must be N^2 message containers between them. If the containers are organized as a square matrix, both the i^{th} row and the i^{th} column are associated with the i^{th} worker. Each row and each column constitutes a *mailbox*. From the

perspective of a worker, its row is its *outbox* and its column is its *inbox*. Thus worker i delivering a message to worker j places the message into container $(i,\ j)$.

However, this causes a read-write conflict. While worker j is reading from its inbox in superstep k, worker i is writing to its outbox messages for superstep $k+1$. These messages sent in superstep k must be received precisely in superstep $k+1$. The solution is to subdivide each container into an *even area* and an *odd area*. In an even superstep, workers write to outboxes in the even area and read from inboxes in the odd area. In an odd superstep the roles are reversed, eliminating the read-write conflict. The synchronization avoidance comes at the expense of space. Each container's size is determined by the number of edges between the workers' partitions. As each container is subdivided, the space requirement is effectively doubled.

We refer to this global organization of message delivery as the *shared mailbox model* and display a diagrammatic overview in figure 5.2.

5.8 Fault Tolerance

Pregel defines a checkpoint-based recover model. A checkpoint interval is given as a number of supersteps. A checkpoint always takes place at the end of a superstep. When a checkpoint is reached, all workers save their state to persistent storage. State includes at least the vertex and edge property maps, and messages received for the next superstep. On failure, the state is recovered from the last checkpoint and the master repartitions and redistributes the workload among the still functioning workers. Execution restarts from there.

The checkpoint interval size is a balance between checkpoint cost and recovery cost. As checkpointing is expensive, Pregel proceeds to defining a *confined recovery* strategy in which in addition to the checkpoints, sent messages are saved for every superstep. The history of failed partitions can then be calculated from the last checkpoint with the history of the remaining partitions.

In C-Pregel, all workers hold their state in Carafe and RStore. Further, at the end of every superstep the state in Carafe and RStore is consistent. Finally, thanks to the shared mailbox model, the inbox of every partition remains unmodified until after the next superstep has started.

It follows that the inbox of a partition for the current superstep can be recovered from RStore at any point in time. At the expense of applying the even-odd area rotation to the remaining state as well, the current superstep is always recoverable by re-assigning and re-executing the failed partitions.

6 C-Pregel Implementation

C-Pregel is divided into multiple submodules. The *master* and *worker* represent the logical components of the system and are both compiled to executable files. They communicate and synchronize via the *metadata manager* module. The latter describes the states and directives of the synchronization state machine in addition to performing the actual communication. The execution of the state machine is in the domain of the master and worker, however. Message passing is delegated to a module called the *message manager*.

Taken together, the implementation is in $\sim$ 2200 lines[1] of templated C++ code.

6.1 Master Life Cycle

The master is the first component to start. On boot it is passed as argument the number of workers in the system. It loads the graph and initializes the metadata manager, which returns an RStore address. This RStore address is the *master address*. With it and the master's RStore namespace, the master can be found by the workers. The master then waits for the workers to boot and be in `waiting` state. It then partitions the graph with the specified partitioning function. The partitions are passed to the workers, and to the initializing message manager. The the workers are also passed the graph's and message manager's locations in RStore. Once that is done, the workers are directed to `initialize`. When all workers are `ready`, the superstep loop is started with the first `go` directive. This continues in a loop as long as workers come back with `done` after completing a superstep. Once all workers are `halted`, the master directs `terminate` and shuts down.

6.2 Worker Life Cycle

The worker takes the master address and namespace as arguments. With these arguments it initializes the metadata manager on boot and signals `waiting` state. When it receives `initialize`, the worker loads the graph and initializes the message manager. From the message manager it gets a reference to the inbox. An instance of the concrete vertex class is instantiated — the concrete vertex type is passed to the worker as template parameter at compile time. The concrete vertex is passed references to the graph and the partition's outbox. For each vertex in its partition, the worker calls `initialize()` on the concrete vertex. At last, the worker sets the current superstep to zero and signals `ready`. On receiving `go`, the worker enters the superstep loop and signals `running`.

Inside the loop, the worker instructs the message manager to fetch messages, and gets an inbox iterator from the inbox. Then the worker begins to sequentially iterate the vertices in its partition using a `ManagedVertexHandle`.

For each vertex, the following happens. The inbox iterator is notified that it is to start a new vertex, so that the iterator can end with the last message in the inbox for the current vertex. A reference to the vertex handle is passed to the concrete vertex object. If the current vertex is active or has new messages waiting, `compute()` is called on the concrete vertex. The method is passed the inbox iterator as argument. When the method returns, two flags in concrete vertex are checked to determine whether the vertex deactivated itself and whether it sent any messages. Then the loop is advanced to the next vertex.

[1] As counted by CLOC v.1.53 (http://cloc.sourceforge.net)

After all vertices are processed, the worker either signals `done` or `halted`, depending on whether or not there are still active vertices in its partition and messages were sent. It then waits on the master's directive. If it is `terminate`, the worker shuts down. Otherwise, if it is `go`, the worker tells the message manager to post the sent messages, increments the superstep, and advances to the next superstep iteration.

6.3 Message Manager

The message manager is an abstraction for the storage of and access to messages. It is supported by two internal submodules, the *mailbox object* and the *inbox iterator*. The mailbox object is a "Proxy"[16, p. 207ff] to an underlying, conceptual mailbox consisting of message areas located in RStore. Depending on whether the mailbox object is being read from or written to, it is called *inbox object* or *outbox object*, respectively. The inbox iterator is a read-only iterator to access the messages within an inbox object.

Note that within this section, the terms "mailbox", "inbox", and "outbox" refer to the design concepts; the mailbox object is referred to verbatim.

Message Storage The type of the messages is defined as template parameter of the concrete vertex. When sent, a message is wrapped in an *envelope* containing the vertex IDs of its source and destination. The vertex name of its source is in the envelope as well, as no high-performance way for translating IDs to names is currently implemented. The message is stored in the designated container area. All areas are allocated by the master on message manager initialization. Each area is physically a fixed-size array, and stored in its own RStore namespace to avoid lock contention in RStore on control path operations. Logically, each area is a queue, to which new messages are appended. The area start and end RStore addresses, capacity, and namespace are stored as metadata in RStore by the message manager.

Mailbox Fetch and Post When the message manager is initialized on a worker, it reads in the RStore addresses and namespaces of all the even and odd areas within the even and odd inboxes and outboxes of the worker.

When the worker requests the inbox to be fetched, the message manager reads the queue metadata of the areas belonging to the inbox of the current superstep from RStore and delegates it to an inbox object. The inbox object maps each such area, reads it as a whole, and sorts its messages as described in the design section. Sorting is implemented using C++ `std::sort()` function, which is a combination of the quicksort, heap sort, and insertion sort algorithms. Then the call returns.

On a post request, the message manager delegates control to the outbox object. The outbox object writes back each area belonging to the outbox of the current superstep to RStore in one go and unmaps it. Then the message manager writes back the queue metadata of the same areas. Finally, the call returns.

Inbox Iteration On creation, the inbox iterator is passed a reference to the worker's inbox object. The first message queue in the inbox is set to be the current queue, and the queue positions of all queues are set to the queue beginnings. When instructed to begin a vertex, the iterator sets the request vertex's ID as the current vertex ID.

When dereferenced, the iterator looks up the position of the current message by means of the current queue and queue position. It returns a reference to the message.

When incremented, the iterator increments the queue position and checks if it is valid. A queue position is valid if it is not past the queue's end, the queue is not empty, and the message at the current queue position is destined for the currently set vertex. If the position is valid, the method returns immediately. Otherwise, the queue list in the inbox is iterated until either

a valid queue position is found or the last queue is reached and is found to have no valid queue position. In both cases, the method returns.

On a comparison, the iterator checks if the current queue position is valid and returns the appropriate Boolean value.

7 Evaluation

In our evaluation of Carafe and C-Pregel, we show that the system scales linearly in the number of workers and linear-logarithmically with graph size. The system performs within 93% of peak performance with minimal caching. It is imbalanced with static workload partitioning, but with better workload models the imbalance is reduced. Finally, we highlight that Carafe's performance is comparable to state-of-the-art graph processing systems in similar settings.

The systems we compare against are:

GraphX GraphX[19] builds on top of Spark, a distributed, data-parallel framework which implements a variant of the MapReduce abstraction. Similar to Carafe, GraphX exposes a set of primitive graph operators, implemented in relational algebra on Spark. On these is set the Pregel-alike graph processing framework we compare against.

GraphLab PowerGraph GraphLab PowerGraph[20] offers a novel gather, apply, scatter (GAS) model. Next to being a model in its own right, GAS is general enough to function as a foundation for other graph processing models, such as the Pregel model. The algorithms we are interested in are implemented directly on the GAS model, however.

We evaluate and compare Carafe with two popular graph algorithms, Dijkstra's algorithm and PageRank. Dijkstra's algorithm is implemented directly on the Carafe API, while PageRank uses our C-Pregel framework. The algorithms are chosen as they represent typical, I/O-bound graph workloads, are simple enough to implement, and are frequently used for comparing graph processing systems.

The tests are performed on a 12 node, IBM x3650 M4 cluster. Each node has two 2.9 GHz, 8-core Intel Xeon E5-2690 CPUs, 256 GB RAM, and three 10 Gb/s, dual-port Chelsio T420-CR RNICs for a combined total of 60 Gb/s. The RNICs support iWARP over Ethernet and are connected with an IBM RackSwitch G8264 switch. With this setup, network latency for an 8 byte RStore read is 9.6μs[3]. The nodes run Debian Linux with a 3.13.11 vanilla kernel.

In the default setting, each cluster node is given one C-Pregel worker for a total of 12 workers; RStore memory servers run on all nodes. The C-Pregel master shares a node with one of the workers, while the RStore master resides on a 13^{th} node outside the core cluster. The setting maximizes bandwidth per worker and minimizes resource contention for measurement consistency.

Parameter	**Setting**
# Workers	12
Partition by	weight
Read caching	enabled
Write caching	disabled
Cache block size	24 MB
Vertex cache size	500 blocks
Edge cache size	1000 blocks

Table 7.1: Carafe Default Parameter Settings

RStore balances chunks across memory servers in a round-robin fashion and is configured to use a 2 MB chunk size. If a read or write operation spans multiple chunks, the chunks will

be spread across multiple RNICs, also in a round-robin fashion. Thus, Carafe's cache block size is configured to be a multiple of ($\#RNICS \times$ *chunk size*). Unless otherwise stated, read caching is enabled, write caching is disabled, and the chosen block size is 24 MB. The cache's default setting is to hold 500 vertex blocks and 1000 edge blocks for a total of 11.7 and 23.4 GB, respectively. This is enough to hold the complete working set data of the graph datasets in cache on a C-Pregel worker.

Graph	Vertices	Edges
LiveJournal	5.4M	79M
Twitter	41M	1.4B
UK 2007/05	106M	3.7B

Table 7.2: Graph Datasets

The graph datasets used in our evaluation are distilled from real-world data. The LiveJournal graph[21] models friend relationships between users on the LiveJournal[1] online social network. The Twitter graph[22] models follower relationships between users on the Twitter[2] online social messaging service. The UK 2007/05 graph[23] is a representation of links between .uk websites as of May 2007.

Each experiment is sampled three times. The presented numbers are the means of the three runs.

7.1 Dijkstra's Algorithm

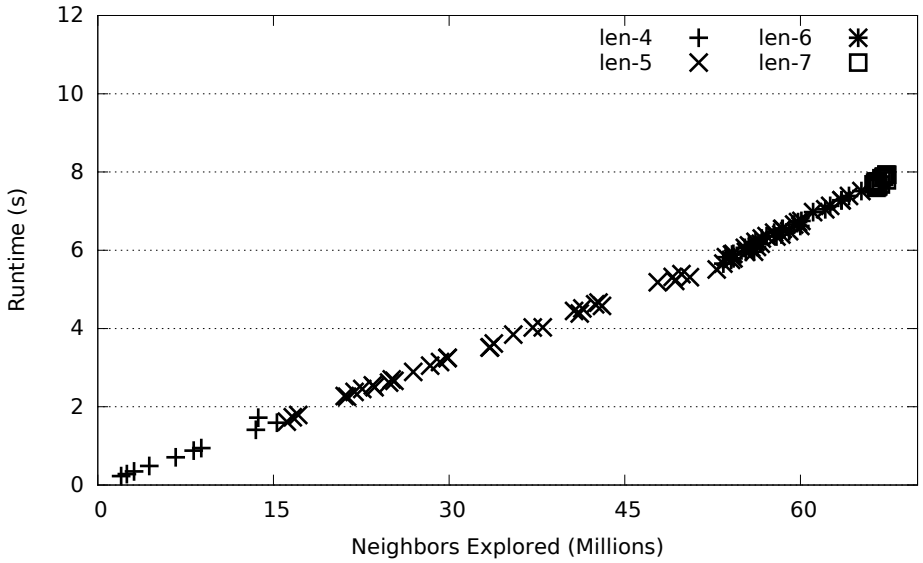


Figure 7.1: Dijkstra's Algorithm Performance

The evaluation of Dijkstra's algorithm's (see section 2.5.1) performance is run on the LiveJournal graph with read and write caching enabled. Caching both reads and writes is necessary to reduce round-trips. Due to the large cache size the entire graph fits into memory, thus the experiment represents best-case performance.

Dijkstra's algorithm has an access pattern as follows: the algorithm's active set is stored in a min-heap data structure in local memory. Each time a vertex is set as current and inspected,

[1] http://www.livejournal.com
[2] http://www.twitter.com

the vertex including its property map is loaded from Carafe. Its adjacency list is scanned sequentially, and each neighbor vertex is loaded from Carafe to compare its distance. If the distance is updated, the vertex is stored to Carafe.

As each load and store represents a data path round-trip to the RStore memory server, Dijkstra is heavily dependent on low latency graph access. The run-time is proportional to the number of neighbors inspected, which has only a very loose correlation to the destination vertex's distance. The correlation is seen clearly in figure 7.1.

Pre-fetching data is possible within an adjacency list, because adjacencies are accessed sequentially. But for the vertices referenced in the adjacency list this is difficult, as vertices are accessed in the order they are listed in the list, effectively random accesses. The same goes for adjacency lists, which is especially detrimental if the lists are short. The next "current" vertex may be located in cache if it was accessed recently in the neighbor inspections. Otherwise, it too is a random access.

Our evaluation of Dijkstra's algorithm is by no means complete; worst-case performance is left to be analyzed, as well as caching behavior. Nevertheless, the experiment shows the effectiveness of the system's zero-copy API and implementation, with no I/O constraints and a challenging memory access pattern.

7.2 PageRank

Our implementation of PageRank (see section 2.5.2) is an identical transcription of the implementation described in Pregel[10]. The only deviations are syntactical adjustments to fit C-Pregel's API. No form of message combining is used, and all PageRank experiments run 32 full iterations.

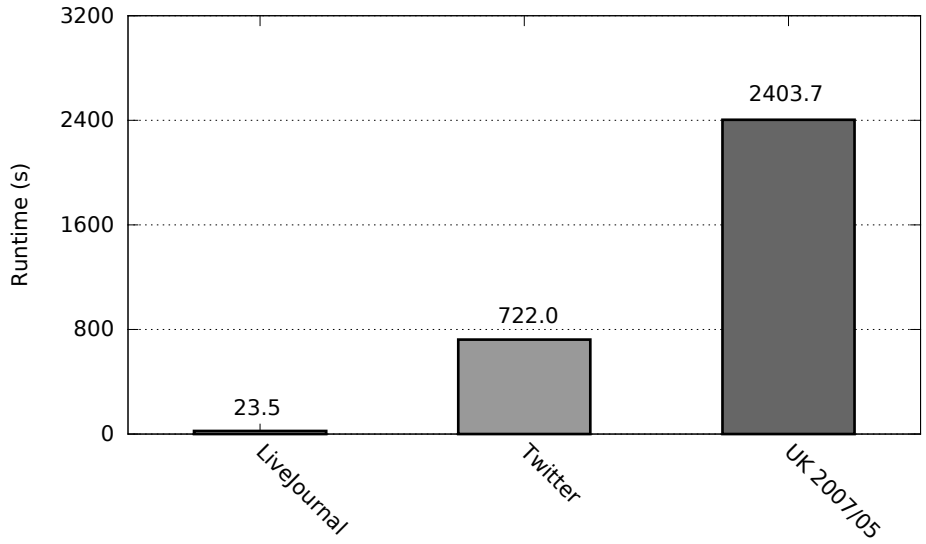
To evaluate PageRank, we first give an overview of how our system performs with the three different graphs, We look at how it scales with the number of workers, and how the cache size affects performance. Then we inspect how skews in workload partitioning are detrimental and where time is spent within a worker. Finally, we compare C-Pregel's performance to that of other systems.

Performance Overview Figure 7.2a depicts the runtime performance of C-Pregel for the PageRank algorithm with the three real-world graphs given as input. The graphs differ in size, with the LiveJournal graph being the smallest, followed by the Twitter and UK 2007/05 graphs. This order of scale is reflected in the comparison, where the runtime is higher for larger graphs. In all of the graphs, the number of edges dominates the number of vertices by a large margin (refer to table 7.2).

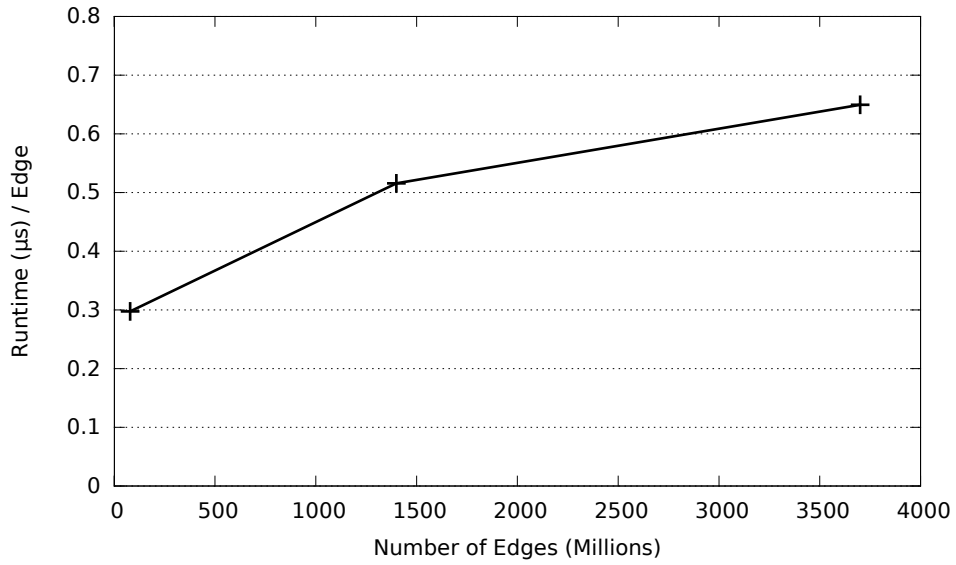
In our implementation of the PageRank algorithm, vertices send an update of their current PageRank to all their neighbors in each superstep, i.e. PageRank performs all-to-all messaging. As C-Pregel does not implement support for message combiners, each such message is passed individually. For the system, this is the worst-case scenario.

Loading vertices and edges from RStore is efficient, as both are accessed sequentially on each worker and can be pre-fetched. After all vertices are processed during a superstep, they are written back to RStore at the latest before the superstep is finished. Like the load, the write-back is sequential and can be performed block-wise. The void of random accesses makes for good cache behavior, meaning accesses are inexpensive time-wise.

Scaling Input Size Putting together the three factors that (a) the number of edges is the dominant in the graphs, (b) there is a message along each edge in every superstep, and (c) loading and storing is efficient, leads us to the hypothesis that passing messages is the dominant cost for PageRank. The hypothesis is supported by comparing the number of edges to the algorithm's runtime in figure 7.2b. The data points suggest that runtime scales linear-logarithmically with



(a) Graph Comparison



(b) Scaling of Runtime w.r.t. the Number of Edges

Figure 7.2: PageRank Runtimes of Graphs

the number of edges, which would reflect the scaling factor of the message passing subsystem's design (see 5.7). However, the data available is insufficient for a thorough analysis of the scaling complexity; more datasets are necessary.

Scaling Workers Scaling the system's size is a determining factor of the success of a distributed system. Figure 7.3 shows the performance of C-Pregel when scaling from 1 to 12 workers in steps of three when processing the LiveJournal graph. The master runs on a dedicated node while the worker count is beneath 12. At twelve workers, the master shares a node with one of the workers. Even while C-Pregel is scaled, the default RStore configuration remains unaltered.

The performance initially starts out with a jump amounting to a 2.3× speedup when going from 1 to 3 workers. Incrementing the number of workers dampens the speedup to 1.7× for each further three-worker step.

The initial speedup is, as expected, nearly equal to the increment in resources. The speedup

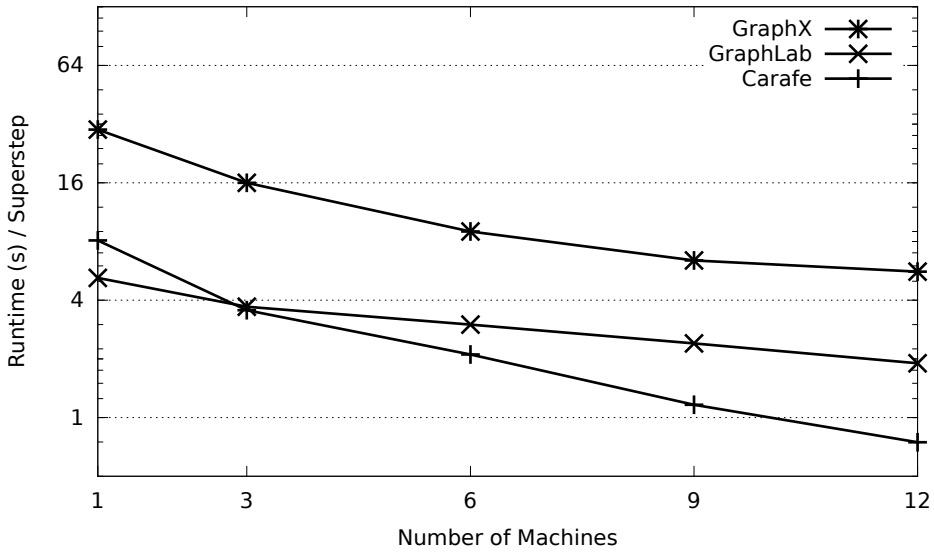


Figure 7.3: Scaling Workers on LiveJournal

factor slows after reaching 3 workers, but is still linear. An explanation for this is that the workload partitioning is not perfect; this alone is not satisfactory, however, as our partitioning comparison below shows. Another contributing factor is that the RStore resources are held constant. Although the bandwidth given to Carafe increases with the number of workers, the bandwidth of the memory servers per worker decreases proportionally to the number of workers. Even so, the total bandwidth of RStore is mostly independent of Carafe's due to the full-duplex network wiring — outbound traffic from a Carafe worker is inbound traffic to an RStore memory server. The direction of traffic is the same across all nodes at a given point in time, as the C-Pregel workers synchronize supersteps and thus are always in approximately the same work phase.

We conclude that bandwidth will become a limiting factor if it is not scaled with the number of workers, although the effect is not yet significant in this experiment.

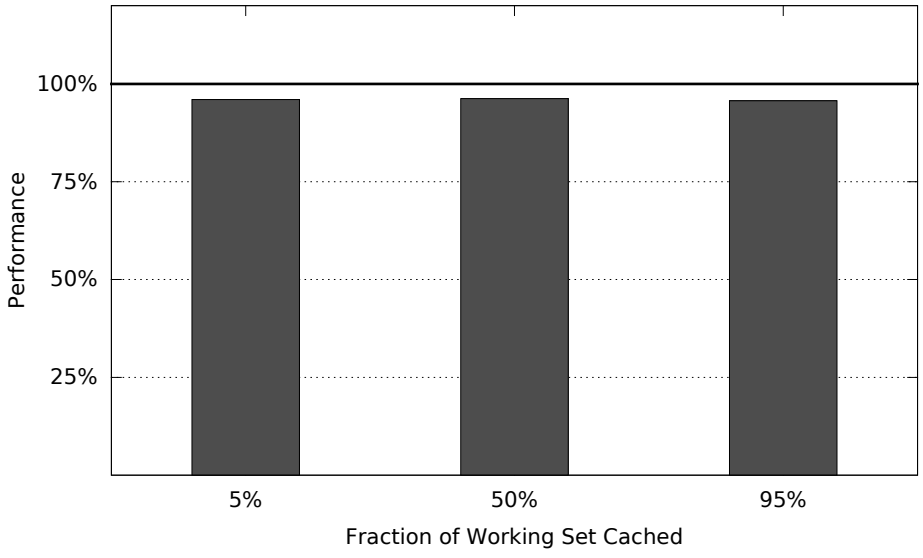


Figure 7.4: Graph Cache Performance

Graph Cache and Pre-Fetching With read caching enabled, part of a worker's working set resides in local memory. We evaluate how minimal and maximal caching perform in combination with the clock eviction strategy (see sections 3.5 and 4.1). To gain insight into why minimal caching behaves as it does, we add more data points in between the two extremes.

The experiment is run on the Twitter graph dataset with the default settings, save for the cache size. Vertices and edges have independent caches; we state cache sizes in the number of blocks as ⟨vertex, edge⟩ tuples.

For minimal caching these are set to one block each, ⟨1, 1⟩ — for the Twitter graph the setting represents 5% of a worker's working set stored in the cache. For maximal caching the default sizes of ⟨500, 1000⟩ blocks are sufficiently large to cache the whole working set on each worker — maximal caching sets the 100% performance baseline. The remaining sample sizes are deduced from the in-RStore, serialized size of the graph. The Twitter graph fits into ⟨93, 467⟩ blocks; if these are distributed evenly among 12 workers, this amounts to ⟨7.75, 38.9⟩ blocks. From these numbers the other two sample sizes are deduced: $\frac{|\text{working set}|}{2}$ and $|\text{working set}| - 1$ amount to ⟨4, 20⟩ and ⟨6, 37⟩ blocks, or 50% and 95% of the working set, respectively. Though the intermediate size settings are not precise because the graph's skewed edge distribution and C-Pregel's by weight work distribution do not partition the graph perfectly, the experiment's conclusion remains unaffected.

The results of the experiment are depicted in figure 7.4. All three settings, minimal caching, ⟨4, 20⟩, and ⟨6, 37⟩ achieve 93% of maximal caching's performance. Although surprising at first, it is easily explained. Access to the graph and property maps is sequential, meaning blocks are pre-fetched. Each block in the working set is loaded once per superstep. In the case of maximal caching, the cache is warmed up in the first superstep, and then the blocks remain available locally. In all other cases, the clock algorithm's LRU eviction degenerates to a first-in, first-out pattern; all blocks must be loaded in every superstep. While the purpose of caching is shattered, pre-fetching amortizes the latency cost of fetching blocks over many graph components. Fast remote graph access reduces the performance penalty for loading blocks on each superstep when processing vertices in PageRank.

If we attribute the difference between minimal and maximal caching to computing vertices (see the worker time profile paragraph below), we calculate a slowdown of 1.27×. We believe this to be a reasonable price for dynamically loading graph data. Cache sizes in-between minimal and maximal caching have no advantage over minimal caching with an LRU-like eviction strategy.

Distribution	Minimum (s)	Maximum (s)
by vertices	4.8	76.7
by outedges	19.7	25.2
by weight	20.6	22.8

Table 7.3: Workload Partitioning Imbalance

Workload Partitioning The target of workload partitioning is that all workers are assigned an equal share of the total workload. In a static scheme, the work is distributed at the beginning of the experiment and henceforth remains unchanged. The difference in the busy time of the least- and most-loaded workers is a measure to determine how well the system is balanced.

Table 7.3 lists the slowest and fastest workers for PageRank on the LiveJournal graph. The rows show the experiment's runtime with each workload partitioning strategy from section 5.6.

Partitioning by vertices is the most simple approach, but the LiveJournal graph's skew in edge distribution results in sub-optimal workload partitioning. When partitioned by outedges, the cost of loading edge blocks and message passing are more equally shared by workers, and the system has somewhat better performance. But the cost of processing vertices is unbalanced.

Merging the two strategies by assigning both vertices and edges a weight of 1 to reflect the costs associated with them yields the desired result of an approximately balanced system.

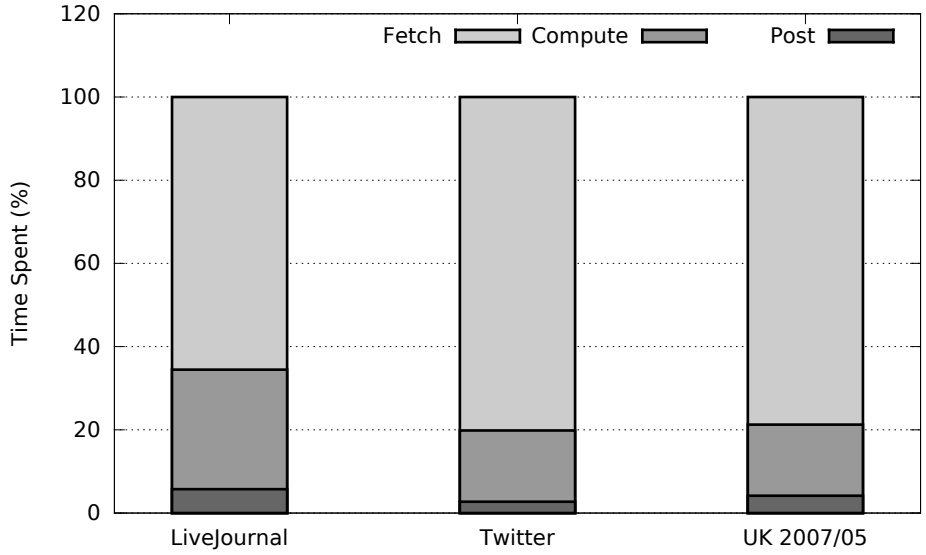


Figure 7.5: Worker Time Profile

Worker Time Profile Breaking down the work time of a worker into the major components shows where time is spent and further optimizations should be concentrated on. Work time is the time effectively spent computing and excludes the time to initialize and synchronize the system. The operations which constitute the main time components are normalized and shown in proportion to the total work time in figure 7.5. The three main work phases — fetching messages, computing vertices, and posting messages — account for almost the complete work time when taken together. Initialization of vertices, setting up and incrementing the vertex and message iterators, and other small operations are at or below the measurement resolution, thus are not included in the figure. The "compute vertex" operation, next to measuring the time to call `compute()` on vertices, encompasses the time to load and store vertex and edge blocks, and to append messages to the send queue; with maximal caching, loading blocks is a one-time cost in the first superstep, but the write-back occurs during each superstep.

The experiment is run on all three graphs with the default settings. In addition to averaging over three runs, the results are also averaged over all 12 workers.

The largest time segment is spent on fetching messages. Posting messages is the smallest contributor, and stands in stark contrast to fetching messages; fetching messages takes between 65% – 80% of the total time. The difference between the two operations is logically small — `read()` is substituted for `write()` — but for one detail: in our implementation fetch sorts messages within the message areas (see section 6.3). Sorting messages accounts for over 90% of the fetch time. Computing vertices with maximal caching is the second-largest time segment, at 17% - 29% of the work time.

As such, the current design has efficient sending and receive iterations at the expense of sorting messages. Although it appears that the time spent to sort messages could be used more effectively elsewhere, the system's goal is to reduce the cost of message passing as much as possible. Being the largest expense in the system, more work is necessary to implement sorting more efficiently and to evaluate other message passing designs.

Comparing Systems The comparison places GraphLab and GraphX into the described 12-machine, single core setting of Carafe. The systems all run native implementations of PageRank.

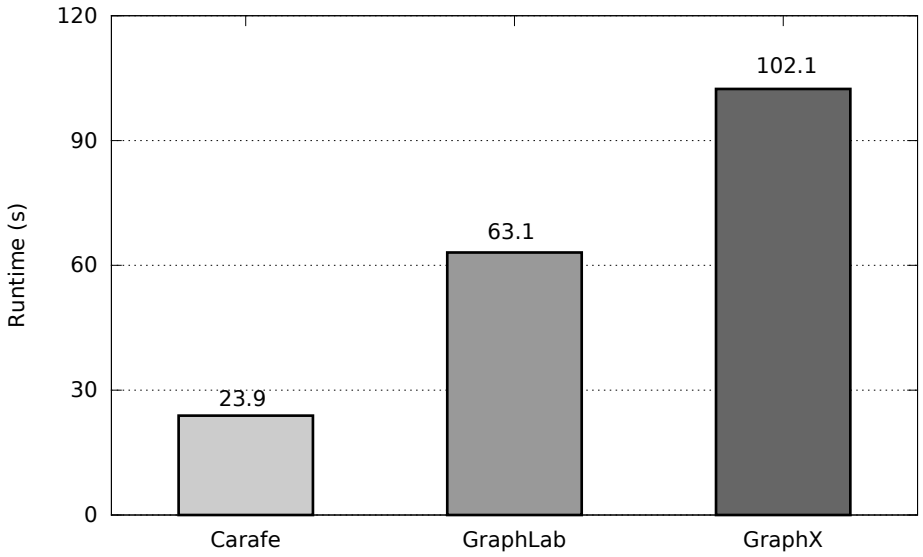


Figure 7.6: Carafe Compared to GraphLab and GraphX

In the case of GraphLab and GraphX the demonstration implementations from their respective code bases are used. As opposed to C-Pregel's implementation, both of these combine messages. The input dataset is the LiveJournal graph.

Our system shows a 2.64× speedup compared to GraphLab PowerGraph 2.2, and a 4.27× speedup compared to GraphX. Given that the other systems show little network activity after the initialization, we deduce that partitioning and combining messages are effective in reducing inter-worker communication. Also, it shows that they, too, perform maximal caching of the graph. Further, the performance numbers roughly correspond to those presented by the authors of GraphX[24] after adjusting for differences in the hardware setup. However, as of this writing, we did not profile the other graph processing frameworks and therefore do not have deep insight into where they spend time.

7.3 Import Format Comparison

In section 4.3 we argue that importing a graph from disk is done often enough to justify converting graphs into a format suitable for fast importing. We then go on to describe an on-disk fast-format and claim that it is more efficiently imported than a raw edge list. The claim is quantified here.

In the experiment the LiveJournal graph is first converted to fast-format using the described conversion tool. Then both fast-format and the edge list are imported into Carafe. Carafe is backed by the default 12-node RStore cluster setup. To ensure that the converter and importers are not bottlenecked by the disk, all input and output datasets are read from and written to Linux's in-memory tmpfs.

As shown in figure 7.7, converting and then importing Carafe's fast-format takes a similar amount of time as importing an edge list. But once the graph is in fast-format, the conversion cost falls away, leaving only the import time. Importing from fast-format takes just 1.68 seconds, a speedup of 37×. As a bonus, the fast-format's binary adjacency list is a factor 1.76× more compact than the ASCII edge list, 0.62 GB versus 1.1 GB.

The reason for the similar performance of conversion and importing of an edge list lies with the underlying application logic — it is virtually the same in both programs. The large speedup when importing from fast-format is due to its optimal structure. The information necessary to allocate RStore resources is pre-calculated and located at the file's beginning. Reading in the

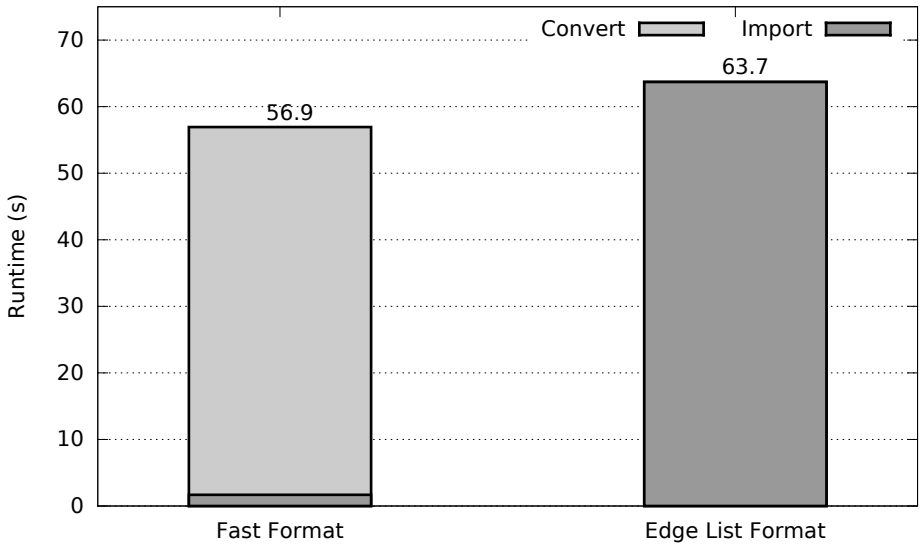


Figure 7.7: Import Format Comparison

graph is then entirely sequential both in memory and on disk, needs only one pass and minimal processing.

When taken together, there is only disk space to be lost to additionally store the graph in fast-format, but much time to be saved.

8 Related Work

This chapter focuses on related work in distributed graph processing. We first present an overview of existing systems and then concentrate on external memory based systems.

Beginnings Initially, there were single-threaded, non-distributed graph processing systems such as the Boost Graph Library (BGL)[25], which provide a useful, graph-oriented abstraction for implementing graph algorithms. BGL comes bundled with a sizeable collection of graph algorithms ready for use. As graph data set sizes grew and multi-core machines became available, a distributed computation model became necessary.

Parallel BGL[26] partially retains the graph-oriented model in a distributed environment. Vertices are distributed to owning nodes, each responsible for its partition during computation. At the partition's edge, "ghost vertices" act as place-holders and caches for adjacent vertices. To take advantage of multiple cores, several of BGL's algorithms are parallelized.

The model's scalability is limited in many senses. Most importantly, edge cuts and hence ghost vertices bring network latency and bandwidth constraints, and the model has no notion of fault tolerance.

Carafe borrows the idea of placing the graph into a shared, globally accessible adjacency list data structure. Similarly to BGL's vertex and edge descriptors, Carafe's vertex handles and iterators abstract the underlying storage structure. But unlike BGL, Carafe does not impose ownership of vertices or otherwise pre-partitions the graph; it is up to the application to control which parts of the graph are mapped. Applications are empowered to fine-grained data operations and to decide if and how graph data is cached. Failures of operations are tolerated by Carafe and the application defines the frequency of write-backs to remote storage.

The Standard Systems Malewicz et. al.[10] argue that the Parallel BGL model does not offer fault tolerance and other features necessary to scale the system. Further, they reason that the MapReduce[27] and SQL models are ill-suited for iterative, stateful graph processing. They propose Pregel, which we describe in detail in section 2.6.1. The model massively parallelizes computation by taking a vertex-oriented view, and reduces the impact of edge-cuts by combining messages. Superstep synchronization and explicit message passing provide opportunity for fault tolerance.

Low et. al.[11] take an alternative path to vertex-centrality: instead of a synchronous, message passing system, Distributed GraphLab implements an asynchronous, shared-memory graph processing model. A choice of consistency models with different trade-offs ensure readers and writers do not conflict. The asynchronous model hides latency induced by storing large graphs in external or remote memory, and, in certain settings, has faster convergence in some algorithms such as PageRank. These benefits come at the expense of a non-deterministic execution model and usability.

The graph structure in Pregel and GraphLab is mostly static. They are unable to string together computations, are limited to a single data set, and employ NP-hard, edge-cut graph partitioning. Combining messages is only possible when there exists an associative and commutative operand to combine them. Nevertheless, they form the current de facto standard in the realm of distributed graph processing systems.

Carafe offers a higher-level, graph-oriented view which is reducible to a vertex-oriented view, as demonstrated with C-Pregel. Defining distributed computation and message passing models is left to the application, Carafe focuses on the graph storage abstraction. The current

implementation also features a static graph structure, but Carafe applications can access an arbitrary number of distinct graph data sets simultaneously and compose multiple computations in sequence.

C-Pregel inherits Pregel's properties, with the differences described in section 5.1. But as opposed to Pregel and GraphLab, C-Pregel does not rely solely on minimizing network traffic for input scalability. Passing individual messages circumvents the associativity and commutativity requirements of message combiners and is feasible also for large graphs. Fault tolerance is inexpensive and has fine resolution. It is to a large degree an integral part of the system, and only a single superstep of a single partition must be recomputed in case of node failure.

Recent Research To avoid the high communication costs associated with edge splits, Gonzalez et. al.[20] exploit the skewed vertex degree distribution of the graph structure to partition by edges. High-degree vertices are split over multiple partitions and the state of these is synchronized. Their PowerGraph framework is a follow-up on GraphLab which introduces a model with gather, apply, and scatter (GAS) phases. The execution engine decides the order in which the phases are run, resulting in either synchronous or asynchronous execution. With these two modes of operation, the Pregel and GraphLab models are shown to be subsets of GAS.

Data-parallel systems have become more universal since the original work on Pregel, and have acquired support for iterative, data-dependent algorithms[28, 29, 30, 31, 32]. With these new armaments comes the power to build efficient graph processing systems on their abstractions. In particular, GraphX[19, 33, 34, 24] by Crankshaw et. al. is a graph processing system based on Spark[31], which casts a set of graph operators to relational algebra. These graph operators are then assembled to higher-level graph abstractions, e.g. the Pregel model. While GraphX's approach has shown competitive performance, its value lies in its novel interpretation of graph processing. Data-parallel operations can be applied to a graph-oriented view, the graph structure being completely mutable. The data-parallel and graph-parallel models are well-integrated, and computations in both models are efficiently pipeline-able. It also does away with limitations to single data sets; GraphX can dynamically combine and split graphs and map data from a database to a graph.

The authors of these frameworks have found a more effective heuristic to partition graphs, and adapted their computational models to fit. They borrow message combining from Pregel and apply it in new ways, for instance PowerGraph's delta-caching, or generalize to a functional approach in GraphX's reduction operators.

Like GraphX, Carafe views the graph as structured data and provides primitive operators to mutate the data. Its more specialized, light-weight nature allows for the introduction of a new concept to graph processing: data and control path separation. Fast access to a remotely stored graph challenges data-locality considerations and changes design considerations. It remains to be seen how this impacts computational flexibility and how modern graph processing models such as GAS benefit from it.

Shao et. al.[35] argue that there is so much diversity in graph structures and algorithms that there exist no generally optimal storage scheme and computation model. To adapt to the diversity, their Trinity system offers a domain-specific language (Trinity Specification Language) to model data and inter-node communication. The TSL automatically generates object-oriented APIs for the data and communication models. Data is physically stored in a distributed, in-memory key-value store. Thus, the graph structure is completely mutable. With their system, they implement the Pregel model.

In spirit, Trinity and Carafe are alike. Trinity and Carafe both separate storage from algorithm, both have distributed, in-memory storage, and both implement Pregel as proof-of-concept of their abstraction. However, we disagree with the vision that a framework must support every possible instantiation of a graph data structure to cope with diversity in graphs and algorithms.

We argue that a handful of well-chosen data structures is sufficient for the vast majority of combinations. With specialization comes potential for optimization. As an example: where Trinity attempts to provide fast graph access with zero-copy serialization, Carafe goes one step further. The local and remote storage structures are identical; the locally mapped parts are merely a window of the whole structure. Combined with explicit data and control path separation, the synchronization of local and remote data is zero-copy end to end.

9 Future Work

The design and implementation efforts have borne fruit, yet there is more work ahead until Carafe is a fully-fledged graph processing system.

9.1 External Vertex and Edge Property Maps

In section 3.4, we discuss the advantages and disadvantages of **external vertex property maps**. In most non-research systems, practical issues often outweigh performance considerations. The time to import a graph repeatedly may well be more than the time gained from higher performance due to locality; indeed we have argued similarly regarding the fast file format. Then again, there may be cases when the external property map design offers higher performance than its internal counterpart, for instance if properties are accessed only infrequently.

Thus far, Carafe does not support **edge property maps**. While they can be simulated with vertex property maps, native support is more space efficient when property maps have fixed size. The arguments for internal versus external property maps are equivalent to their vertex pendants. Since edge properties are often defined in lieu with the graph structure as opposed to being initialized to a constant or structure-related value, the importers must be modified accordingly.

For both vertex and edge property maps, **byte-wise granular read and write** should be supported. Although their usefulness is greatly diminished when read or write caching is enabled, in some situations they are essential. For example, imagine two workers concurrently accessing disjunct properties within the same property map. If the whole map is read and written back by both workers, depending on the serialized order of operations, one worker will overwrite changes made by the other worker. The workers can avoid overwriting changes by mutating their fields with finer granularity.

9.2 Dynamic Scheduling

Dynamic scheduling of the workload in C-Pregel is challenging in a system with resource pre-allocation at its core. As we have shown, a well-tuned static scheduler can achieve good performance. But static scheduling only goes so far. It requires hand-tuning of parameters to the algorithm and computational environment for optimal performance. Diversity in these factors makes finding optimal, static parameters for all settings impossible. To make matters even worse, stragglers during processing of large graphs can prolong the runtime significantly while wasting resources. There are three approaches to explore which do not compromise the current system's design.

Neighbor Shifting Firstly, given the contiguous vertex and adjacency list arrays, it is desirable to keep the workload partitions contiguous as well, both to avoid unnecessary complexity and to optimize block caching. Secondly, the matrix of message areas is inflexible; arbitrary workload re-allocations would require re-allocation of the affected areas to move memory capacity from one RStore namespace to the other.

The first point is solved if vertices are rebalanced only between neighboring workers, where workers are neighbors if the conjunction of their partitions has contiguous vertex IDs. Thus, one worker shifts part of its workload to its neighbor.

The second point can be fixed by unifying the areas into a single namespace and allocating them contiguously. Space in the message slots could then be shifted from one message area to another by logical reallocation. As the donor area would shrink while the recipient area would grow, the areas in-between would need to be shifted and their messages moved (i.e. copied). The RStore addresses for the message areas would be reset to the new addresses and sizes on each worker.

This whole process requires the complete system to be at a stand-still (stop-the-world). An opportune moment for rebalancing is in-between supersteps.

Intra-Worker Work Stealing Within a worker, threads share the graph's block cache and have fast access to the in-memory state of other threads. Work stealing[36] then functions as follows. Each thread stores the vertex IDs of its unprocessed workload in a queue. A thread works through its own queue by popping work units off the front of the queue. Now, if a thread's queue is empty, meaning it has run out of work, it attempts to pop work units off the back of another queue and pushes them onto its own queue; the victim queue is selected uniformly at random. The concurrent queue access implies that the push, pop-front, and pop-back operations on a queue must be thread-safe.

The remaining issue to resolve is the inbox iterator. If access to messages is not sequential, the benefits of the design are lost. However, if work is stolen in contiguous chunks, the victim's inbox iterator's end could be adjusted, and the thief would create a new inbox iterator for his loot.

The drawback to this design is that vast parts of the message passing subsystem must be made thread-safe. Next to complexity, this adds time to acquire locks or perform atomic operations even when there is no contention.

Micro-Partitions If there are N workers in the system, the graph is divided into N partitions. Currently, each worker is assigned exactly one partition. Dividing the graph into smaller *micro-partitions*, each worker initially receives more than one partition. Micro-partitions can then be re-assigned to balance the workers' workloads.

Micro-partitioning is the most simple of the presented methods to implement. The drawback is that large partitions only allow coarse-grained balancing, while very small partitions increase the overhead of managing them and reduce the performance benefits of sequential access.

These methods can all be combined and matched to play their strengths and overcome their individual weaknesses. For instance, micro-partitions could coarsely balance work between workers, and within each worker work stealing would finely distribute the workload among the threads. The combination would overcome the limitations of static scheduling, making Carafe a more practically usable system.

9.3 Multi-Threading

Modern *symmetric multi-processing (SMP)* machines have many cores and large amounts of main memory. They have a layered cache hierarchy and *non-uniform memory access (NUMA)*. To use them to their full capacity, a distributed system such as C-Pregel has two possibilities: processes and threads.

Multiple, single-threaded processes, as in the current design, can run on the same machine to maximize resource utilization. They are oblivious of their close locality and communicate via the network.

In contrast, a single, multi-threaded process is aware of its threads' close locality. In the Pregel model there is no immediate benefit in sharing the graph cache, because the workloads of the individual threads are disjoint. However, instead of exchanging messages over RStore,

passing a pointer to a message area is sufficient as the threads share the same address space. This saves bandwidth and is orders of magnitude faster. A second potential benefit from the shared address space comes in the form of workload partitioning (see section 9.2 above).

Design Draft The master and worker need alteration to accommodate threads. The worker delegates its work loop to the threads. Each thread receives its own mailboxes, which in an initial implementation remain mostly unmodified from their current design. The worker's block cache must be made thread-safe for concurrent accesses in the case of block sharing. The worker's main thread is promoted to locally managing an extended state machine. Lastly, the core state machine between the master and worker remains unchanged, but is extended such that the worker checks the state of all workers before signaling the master and propagates directives to its threads. All the while, the correctness requirements of the state machine must be kept intact.

Workload Partitioning On startup, each worker informs the master about its number of threads so that the master has a global view of resources. The master can then split the graph into as many partitions as there are threads and distribute the workload directly to each thread. With a global picture and fine-grained control of resources, the master balances workload not only in homogeneous, but also in inhomogeneous clusters.

10 Conclusion

In this work we have demonstrated that graph storage and computation are effectively and efficiently separable in a distributed setting. Carafe illustrates how to design and implement a distributed, in-memory graph storage abstraction layer as a foundation for high-performance, distributed graph processing systems. Our framework uses RStore as basis, a general-purpose data store that leverages RDMA to deliver high-bandwidth, low-latency access to data stored in remote DRAM. By treating all graph data as remote but readily accessible, our system implementation demonstrates how to translate high performance at the network layer into high performance at the graph application layer. Carafe's capabilities are shown by way of C-Pregel, a Pregel-alike, distributed graph processing system, and an implementation of Dijkstra's single source shortest path algorithm.

Together, Carafe and C-Pregel are competitive in performance with state-of-the-art systems in calculating PageRank. The system scales linearly to linear-logarithmically with input size, and linearly with network and CPU resources. With Carafe we quantify the performance impact of dynamically loading graph data from remote storage in comparison to exclusively relying on caches, and conclude that, in our setting, dynamic loading is feasible for large graphs. Likewise, C-Pregel opens the doorway to graph computations where optimization based on graph partitioning and mathematical properties is difficult.

Bibliography

[1] J. Hilland, P. Culley, J. Pinkerton, and R. Recio, "Rdma protocol verbs specification," IETF, Tech. Rep. 1.0 (Draft), 2003.

[2] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia, "A remote direct memory access protocol specification," IETF, RFC 5040, 2007.

[3] A. Trivedi, P. Stuedi, B. Metzler, M. Schmatz, and T. R. Gross, "RStore: A direct-access dram-based data store," IBM Research, Technical Report RZ3879, 2014.

[4] M. E. Newman, "Power laws, pareto distributions and zipf's law," *Contemporary physics*, vol. 46, no. 5, pp. 323–351, 2005.

[5] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 4. ACM, 1999, pp. 251–262.

[6] A. Clauset, C. R. Shalizi, and M. E. Newman, "Power-law distributions in empirical data," *SIAM review*, vol. 51, no. 4, pp. 661–703, 2009.

[7] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[8] K. Mehlhorn and P. Sanders, *Algorithms and data structures: The basic toolbox*. Springer, 2008.

[9] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. ACM, 2010, pp. 135–146.

[11] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[12] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990. [Online]. Available: http://doi.acm.org/10.1145/79173.79181

[13] C++ Standards Committee and others, "Standard for programming language C++," ISO/IEC, Tech. Rep. 14882:2011, 2011.

[14] R. Bryant and D. R. O'Hallaron, *Computer systems: a programmer's perspective*. Prentice Hall, 2003.

[15] F. J. Corbato, "A paging experiment with the multics system," DTIC Document, Tech. Rep., 1968.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[17] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX Annual Technical Conference*, vol. 8, 2010, p. 9.

[18] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[19] D. Crankshaw, A. Dave, R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "The graphx graph processing system."

[20] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *OSDI*, vol. 12, no. 1, 2012, p. 2.

[21] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006, pp. 44–54.

[22] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW '10: Proceedings of the 19th international conference on World wide web*. New York, NY, USA: ACM, 2010, pp. 591–600.

[23] P. Boldi, M. Santini, and S. Vigna, "A large time-aware graph," *SIGIR Forum*, vol. 42, no. 2, pp. 33–38, 2008.

[24] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX, 2014, pp. 599–614.

[25] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *Boost Graph Library: User Guide and Reference Manual, The*. Pearson Education, 2001.

[26] D. Gregor and A. Lumsdaine, "The parallel bgl: A generic library for distributed graph computations," *Parallel Object-Oriented Scientific Computing (POOSC)*, p. 2, 2005.

[27] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[28] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[29] ——, "The haloop approach to large-scale iterative data analysis," *The VLDB Journal — The International Journal on Very Large Data Bases*, vol. 21, no. 2, pp. 169–190, 2012.

[30] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "Ciel: A universal execution engine for distributed data-flow computing." in *NSDI*, vol. 11, 2011, pp. 9–9.

[31] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 10–10.

[32] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[33] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2013, p. 2.

[34] R. S. Xin, D. Crankshaw, A. Dave, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: Unifying data-parallel and graph-parallel analytics," *arXiv preprint arXiv:1402.2394*, 2014.

[35] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 505–516. [Online]. Available: http://doi.acm.org/10.1145/2463676.2467799

[36] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55 – 69, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731596901070