Coflow: A Networking Abstraction for Cluster Applications

Mosharaf Chowdhury, Ion Stoica
EECS, UC Berkeley, CA, USA
{*mosharaf, istoica*}*@cs.berkeley.edu*

ABSTRACT

Cluster computing applications – frameworks like MapReduce and user-facing applications like search platforms – have application-level requirements and higher-level abstractions to express them. However, there exists no networking abstraction that can take advantage of the rich semantics readily available from these data parallel applications.

We propose *coflow*, a networking abstraction to express the communication requirements of prevalent data parallel programming paradigms. Coflows make it easier for the applications to convey their communication semantics to the network, which in turn enables the network to better optimize common communication patterns.

Categories and Subject Descriptors

C.2 [**Computer-Communication Networks**]: Distributed Systems—*Cloud Computing*

General Terms

Design

Keywords

Cluster Networking, Coflow, Data-Intensive Applications, Datacenter Networks, Cloud Computing

1 Introduction

Cluster computing applications serve diverse computing requirements, and they expect a broad spectrum of services from the network. On the one hand, some of these applications are throughput-sensitive; they must finish as fast as possible and must process every piece of input (e.g., MapReduce [13], Dryad [17]). On the other hand, some are latency-sensitive with strict deadlines, but they might not require exact answers (e.g., search results from Google or Bing, home feed in Facebook). A large body of work has emerged to address the communication requirements of cluster computing applications [6, 11, 15, 16, 26, 29, 30].

Unfortunately, the networking literature does not provide any construct to express the communication requirements of datacenter-scale applications. Specifically, the abstraction of flows cannot capture the semantics of communication between two groups of machines in a cluster application, where the collective fate of all the flows between the two groups is more important than that of any individual flow. The lack of an abstraction has several consequences. First, it promotes point solutions with limited applicability. Second, it can result in solutions that have not been optimized for appropriate objectives. Finally, without an abstraction, it is hard to reason about the underlying principles and to anticipate problems that might arise in the future.

In this paper, we study prevalent parallelization models for cluster computing (e.g., dataflows with/without barriers, Bulk Synchronous Parallel, and partition-aggregate) and their communication requirements (§2). We observe that most of these applications are organized into multiple stages or have machines grouped by functionality. Communication takes place at the level of machine groups, and it is often dictated by application-specific semantics.

Based on our observations, we propose *coflow*, a networking abstraction that captures diverse communication patterns observed in cluster computing applications (§3). Each coflow is a collection of flows between two groups of machines with associated semantics and a collective objective. The semantics allow the network to take different actions on the collection to achieve the mutual end goal. Several existing proposals have used the notion of such collections or hinted at it, albeit in limited scopes [11, 16, 26]. In this paper, we consider the amount of semantic information necessary to take advantage of such collections (§3). We propose an intent-driven API for cluster applications to convey the required semantic information to the network (§4). Finally, we discuss how the coflow API enables innovation in cluster applications and the network by decoupling application intents from underlying mechanisms (§5).

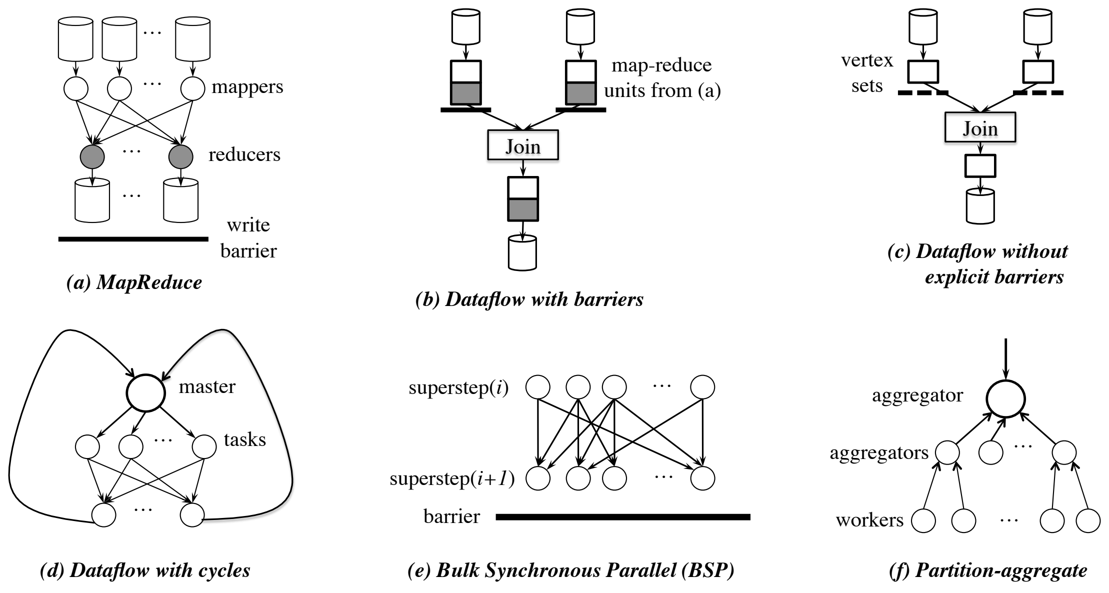


Figure 1: Communication patterns in cluster computing applications: (a) Shuffle and DFS replication in MapReduce [13]; (b) Shuffle across multiple MapReduce jobs in dataflow pipelines that use MapReduce as the building block (e.g., Hive [5]); (c) Dataflow pipelines without explicit barriers (e.g., Dryad [17]); (d) Dataflow with cycles and broadcast support (e.g., Spark [28]); (e) Bulk Synchronous Parallel or BSP (e.g., Pregel [19]); (f) Aggregation during partition-aggregate communication in online services (e.g., user-facing backend of search engines and social networks).

2 Communication in Cluster Applications

Most cluster computing applications are frameworks (e.g., MapReduce) that take user-defined jobs and follow specific workflows enabled by corresponding programming models. Some others are user-facing pipelines, where user requests go through a multi-stage architecture to eventually send back corresponding responses (e.g., search results from Google or Bing, home feed in Facebook). In this section, we study prevalent cluster computing applications and their communication patterns (Figure 1), compare their requirements, and summarize existing solutions that address these requirements.

2.1 MapReduce

MapReduce [2, 13] is a well-known and widely used cluster computing framework. In this model, each mapper reads its input from the distributed file system (DFS) [4,14], performs user-defined computations, and writes intermediate data to the disk; each reducer pulls intermediate data from different mappers, merges them, and writes its output to the DFS, which then replicates it to multiple destinations.

Given m mappers and r reducers, a MapReduce job will create a total of mr flows for shuffle, the process of transferring intermediate data, and at least r flows for output replication. The primary characteristic of communication in the MapReduce model is that a job will not finish until its last reducer has finished. Consequently, there is an explicit barrier at the end of the job, which researchers have exploited for optimizing shuffles in this model [11]. Similar optimizations are applicable to cross-rack replication of DFS writes as well, because all tasks must finish writing for a job to finish writing its output.

2.2 Dataflow Pipelines

While MapReduce is very popular, it is not the most expressive of data parallel frameworks. There exist a collection of dataflow pipelines that address many deficiencies of MapReduce, and they have diverse communication characteristics.

Dataflow with Barriers A dataflow pipeline with multiple stages uses MapReduce as its building block (e.g., Sawzall [22], Pig [21], Hive [5]). Consequently, there are barriers at the end of each building block, and this paradigm is no different than MapReduce in terms of communication.

Dataflow without Explicit Barriers Some dataflow pipelines do not have explicit barriers and enable higher-level optimizations of the operators (e.g., Dryad [17], DryadLINQ [27], SCOPE [9], FlumeJava [10], MapReduce Online [12]); a stage can start as soon as some input is available. Because there is no explicit barrier, barrier-synchronized optimization techniques are not useful. Instead, researchers have focused on understanding the internals of the communication and optimizing for specific scenarios [15, 30].

Dataflow with Cycles Traditional dataflow pipelines unroll loops to support iterative computation requirements. Spark [28] obviates loop unrolling by keeping in-memory states across iterations. However, implicit barriers at the end of each iteration allow MapReduce-like communication optimizations in cyclic dataflows [11]. These frameworks also

Model	Examples	Barrier Sync.?	Barrier Type	Loss Tolerance	Comm. Objective
MapReduce	[2,13]	Yes	Write to the DFS	None	Minimize completion time
Dataflow with Barriers	[5,21,22]	Yes	Write to the DFS	None	Minimize completion time
Dataflow w/o Explicit Barriers	[9,10,12,17,27]	Yes (Implicit)	Input not ready	None	Minimize completion time
Dataflow with Cycles	[28]	Yes (Implicit)	End of iteration	None	Minimize completion time
Bulk Synchronous Parallel	[1,3,19]	Yes	End of superstep	None	Minimize completion time
Frameworks w/o Barrier Synchronization	[18]	No	None	App. Dependent	Either
Partition-Aggregate	Search/Social	Yes	End of deadline	App. Dependent	Meet deadline

Table 1: Summary of communication requirements in popular cluster computing applications.

provide communication primitives like broadcast and many-to-one aggregation that, unlike shuffle, push data to a set of already known destinations.

2.3 Bulk Synchronous Parallel (BSP)

Bulk Synchronous Parallel or BSP is another well-known model in cluster computing. Examples of frameworks using this model include Pregel [19], Giraph [1], and Hama [3] that focus on graph processing, matrix computation, and network algorithms. A BSP computation proceeds in a series of global supersteps, each containing three ordered stages: concurrent computation, communication between processes, and barrier synchronization. With explicit barriers at the end of each superstep, the communication stage can be globally optimized for the superstep.

2.4 Frameworks without Barrier Synchronization

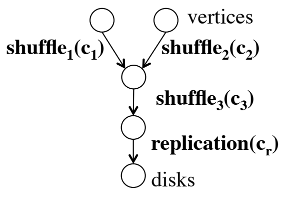
Sometimes complete information is not needed for reasonably good results; iterations can proceed with partial results. GraphLab [18] is such a framework for machine learning and data mining on graphs. Unlike BSP supersteps, iterations can proceed with whatever information is available as long as it converging; missing information can asynchronously arrive later.

2.5 Partition-Aggregate

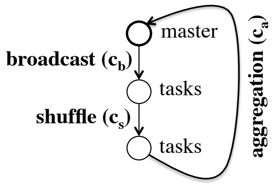
User-facing online services (e.g., search results in Google or Bing, home feeds in Facebook) receive requests from users and send it downward to the workers using an aggregation tree. At each level of the tree, individual requests generate activities in different partitions. Ultimately, worker responses are aggregated and sent back to the user within strict deadlines. Responses that cannot make it within the deadline are either left behind [26] or sent later asynchronously (e.g., Facebook home feed). Research in this direction have looked at dropping flows [26], preemption [16], and cutting long tails [29]; however, they do not exploit any application-level information.

2.6 Summary of Communication Requirements

Despite differences in programming models and execution mechanisms, most cluster computing applications have one thing in common: they run on a large number of machines that are organized into multiple stages or grouped by functionality [8]. Each of these groups communicate between themselves using a few common patterns (e.g., shuffle, broadcast, and aggregation) with two primary objectives: minimizing completion times and meeting deadlines.



(a) Dryad



(b) Spark

Figure 2: Graphical representations of the applications in Figures 1(c) and 1(d) using coflows. Circles represent machine groups (M) and edges represent coflows (C).

Table 1 summarizes the key characteristics of the aforementioned cluster computing applications in terms of the presence of synchronization barriers, characteristics of such barriers, the ability of the application to withstand loss or delay, and the primary objective of communication.

3 Coflow

Even though individual flows are indistinguishable at the transport layer, flows between groups of machines in a cluster computing application often have application-level semantics. For example, the last flow in an mr-shuffle determines its completion time [11]. Similarly, if one decides to delay or drop flows that will miss their deadlines [26], one might want to restrict these actions to a small number of requests. In this section, we introduce the notion of a coflow that captures the semantics of a collection of flows, and we discuss the interactions between multiple coflows.

3.1 What is (in) a Coflow?

We refer to a semantically-related collection of flows between two groups of machines as a *coflow*. Each coflow contains information about its structure and the collective objective of its flows.

Structure Each coflow $c(S,D) = \{f_1, f_2, \ldots, f_{|c|}\}$ is a collection of flows (f_i) between machines in machine groups S and D, where $|c|$ denotes the cardinality (i.e., the number of flows) in c. Machines in S and D may or may not overlap. For example, a coflow representing an mr-shuffle in a MapReduce job will have m mappers in S, r reducers in D, and a total of mr flows (i.e., $|c| = mr$).

We can now represent a cluster computing application using a graph $G = (M, C)$, where M is the set of its machine groups that are connected by coflows in C, the set of coflows. Figure 2 depicts the Dryad and Spark jobs in Figure 1(c) and Figure 1(d) using four different coflow patterns.

If we denote the start and end times of each flow $f_i \in c$ by $start(f_i)$ and $end(f_i)$, the start and end times of the coflow can be represented by $start(c) = \min_{f_i} start(f_i)$ and $end(c) = \max_{f_i} end(f_i)$. The duration or the completion time ($\overline{c}$) of c becomes $\overline{c} = end(c) - start(c)$. We denote the total number of bytes transferred using f_i by $size(f_i)$.

Objective The collective objective of a coflow not only dictates the joint optimization that can be performed on its flows, but also determines the necessary pieces of information required for that optimization. Consider the two predominant objectives in cluster communication:

- *Meet deadline* ($\overline{c} < \mathbf{D}$)*:* In order to finish within a deadline $\mathbf{D}$, one can set rates for all flows to finish at $\mathbf{D}$ [26].
- *Minimize completion time (*minimize $\overline{c}$*):* One way to achieve this is to set the rates of individual flows such that the slowest progressing one (f_{last}) finishes as fast as possible, and the rest finishes by $end(f_{last})$ [11].

Both objectives require knowledge of $size(f_i)$ for all $f_i \in c$. What action(s) must be taken, e.g., if a deadline is missed, depends on the application and might require additional information.

Availability of Information Even in large clusters, occasionally an application cannot perform all its computation (within and across stages) in parallel. Consequently, when a coflow starts, all of its flows might not be active; characteristics of some flows might be completely unknown! During a shuffle, for example, a mapper can create intermediate data for a reducer that is yet to start (unknown destination); similarly, a reducer can wait on a mapper to run and create intermediate data (unknown source and flow size). The cardinality of a coflow is typically known.

However, the availability of information varies from one application to another. For example, in long running services, the sources as well as the destinations of each flow in an aggregation coflow are known; the maximum size of individual flows are known as well.

In either scenario, the objective of a coflow is known when the coflow is created, and it does not change over time. The (un)availability of information has several implications:

- *Time and space decoupling:* Endpoints of individual flows in a coflow can be decoupled in time, meaning senders and corresponding receivers might not be active simultaneously. This requires storage in the network. Mappers writing their outputs to local disks approximate time decoupling during shuffle in MapReduce.
  Flows can be decoupled in space as well. For example, a recipient can receive a broadcasted piece of data from the originating source as well as from a recipient that has received it already.
- *Push-vs-pull semantics:* When the destination of a flow is known, data can be actively pushed to the destination (e.g., in dataflow pipelines w/o barriers [17] and during Spark broadcasts [11]). On the contrary, with unknown destinations, receivers must pull from different sources.

3.2 Interactions Between Coflows

In a shared cluster, multiple coflows from one or more applications can be active in parallel. When creating a new coflow, its interactions with the existing coflows can be expressed using the following concepts.

Sharing Sharing the cluster network among multiple coflows is an active area of research [7, 23, 24]. Given two concurrent coflows c_1 and c_2 from two different applications with demands $\mathcal{D}(c_1)$ and $\mathcal{D}(c_2)$, we consider how to express their allocations $\mathcal{A}(c_1)$ and $\mathcal{A}(c_2)$ of the shared network $\mathcal{N}$.

Reservation schemes [7] are the easiest ones to articulate: each coflow gets whatever fraction of $\mathcal{N}$ they asked/paid for. One can also ensure max-min fairness between c_1 and c_2 over $\mathcal{N}$ by progressively filling their demands, $\mathcal{D}(c_1)$ and $\mathcal{D}(c_2)$. Finally, to provide network proportionality [23], one needs to maintain the invariant $\mathcal{A}(c_1)/\mathcal{A}(c_2) = |c_1|/|c_2|$.

Prioritization While frameworks allow assigning priorities to individual jobs, they cannot ensure these priorities in the network. By using priorities as weights, one can provide larger allocations to coflows from applications with higher priorities, i.e., $\mathcal{A}(c_1)/\mathcal{A}(c_2) = P(c_1)\mathcal{D}(c_1)/P(c_2)\mathcal{D}(c_2)$, where $P(.)$ denotes the priority function.

Ordering All coflows belonging to the same application have the same priority. Often, however, there exists an implicit ordering of coflows within an application. Consider the Dryad application in Figure 2(a), and assume that shuffle$_i$ is denoted by c_i. Because Dryad does not introduce explicit barriers between its stages, c_3 can start before either c_1 or c_2 has finished. However, the progress of c_3 depends on the progress of its predecessors, and both c_1 and c_2 must *finish before* c_3. We denote the "finishes before" relationship between the coflows by $c_1, c_2 \geq c_3$, where concurrent execution of c_3 alongside c_1 and c_2 is expected. Now assume that the cross-rack replication of job output (c_r) should *start after* c_3 has finished. We denote the "starts after" relationship between c_r and c_3 by $c_3 > c_r$.

In Figure 2(b), the broadcast in iteration i must start after the aggregation from iteration $(i-1)$ has finished ($c_a(i-1) > c_b(i)$); within the i^{th} iteration, the broadcast must finish before the shuffle ($c_b(i) \geq c_s(i)$), and the shuffle before the aggregation ($c_s(i) \geq c_a(i)$).

4 The Coflow API

We propose an intent-driven API [25] for coflows that hides the underlying mechanism of communication and allows an

Operation	Caller
create$(pattern, [options]) \Longrightarrow handle$	Driver
update$(handle, [options]) \Longrightarrow result$	Driver
put$(handle, id, content, [options]) \Longrightarrow result$	Sender
get$(handle, id, [options]) \Longrightarrow content$	Receiver
terminate$(handle, [options]) \Longrightarrow result$	Driver

Table 2: Coflow API operations and calling entities.

application to provide as much semantic information as necessary.

We identify four entities that are involved in coflow API invocations: the *driver* that coordinates a cluster application and its coflows, *senders* and *receivers* of individual flows in a coflow, and the *network* that performs the actual communication based on coflow characteristics.

The coflow API provides five high-level operations (Table 2). Drivers initialize a coflow through **create()**, which creates a private namespace for the coflow and returns a unique coflow $handle$. The $pattern$ of a coflow, such as *shuffle*, *broadcast*, or *aggregation*, determines its default behavior (e.g., broadcast supports space decoupling). Additional information can be provided and existing behavior can be overridden through $options$, which is an optional list of key-value pairs. Once a coflow has been created, drivers can change coflow characteristics using the **update()** operation. The priority of a coflow and its dependencies on others are created and updated using these two operations as well.

A sender expresses its intent to insert a $content$ with an identifier id into the network using **put()**. For example, a mapper would **put()** r pieces of $content$ for r reducers in a MapReduce job. The id of a $content$ is unique within a coflow namespace. Any flow created to transfer this piece of data belongs to the coflow with the specified $handle$. The recipient(s) of $content$, if known, can be passed along through $options$; the network determines whether and when to actually transfer the data. Receivers indicate their interest in retrieving a content using its id through **get()**. The network determines when and from where to retrieve the requested piece of data based on coflow characteristics. Exact implementations of the **put()** and **get()** operations can be based on transfer plugins as explained in [25].

The **terminate()** call from the driver signals the completion of a coflow, and the network can safely release resources dedicated to it.

5 Using Coflows

In this section, we discuss how coflows make it simpler to design cluster applications and how the coflow API provides flexibility in designing the underlying network and to optimize frequently used coflows.

5.1 Developing Cluster Applications with Coflows

Writing a brand new cluster computing application or extending an existing one to accommodate new communication requirements boils down to specifying the required coflows – their structures, objectives, and corresponding priorities and ordering – using the coflow API.

For an example, consider extending MapReduce to support broadcast.[1] Assume that each mapper receives a common piece of data from the driver in addition to its input. The driver initializes the broadcast to get a handle for the coflow and inserts the content into the network:

$handle \leftarrow$ **initiate**$(broadcast)$

put$(handle, id, content)$

Each mapper receives the coflow handle from the driver and uses it to retrieve the broadcasted content:

get$(handle, id)$

Once all the mappers have finished, the driver terminates the broadcast:

terminate$(handle)$

The actual distribution of data due to **put()** and **get()** is handled or dictated by the network depending on its design.

5.2 Designing the Network

Since the coflow API expresses the intent of an application without invoking any specific mechanism, it provides flexibility in designing the underlying network in terms of naming, addressing, and content delivery mechanisms.

Applications will call the coflow API and offload their coflows to a distributed middleware. The middleware control plane will periodically determine the network shares of individual coflows given their objectives by taking relative priorities and ordering into account, and its data plane will transfer data using given shares of the network.

To enable time decoupling, data must be (temporarily) stored until it is retrieved by the intended receiver(s). A separate middleware can be used for storage with appropriate plugins to write $content$ to and read it from the storage medium [25].

While existing applications must be patched to use the coflow API, host OSes and hypervisors can be used without any modification.

5.3 Optimizing Common Coflows

A handful of coflow structures (e.g., one-to-many, many-to-one, and many-to-many) can satisfy most communication requirements across a wide spectrum of cluster applications. This calls for standardizing, optimizing, and making the prevalent coflows available for reuse, which will ensure less bugs and better overall performance. Table 3 presents possible ways to implement various coflows by setting rates of individual flows and through flow prioritization.

5.4 Extending Coflows

While researchers have made several proposals in recent years to ensure that (co)flows meet their deadlines [16, 26, 29], most coflows can be expressed using only two objectives (Table 1). Similarly, there exist several proposals on sharing

[1]MapReduce does not have native support for broadcast, which hurts many iterative applications [11, 28]. Examples include applications using the Expectation-maximization (EM) algorithm.

Coflow	Mechanism to Achieve the Coflow Objective
Shuffle	Set rates of individual shuffle flows such that the slowest one finishes as fast as possible [11].
DFS Replication	Set rates of individual replication flows such that the slowest one finishes as fast as possible.
Broadcast	Allocate rates at broadcast participants so that the slowest receiver finishes as fast as possible [11].
Comm. in Dataflows w/o Exp. Barriers	Set rates/prioritize so that later stages do not block on this coflow.
BSP Communication	Set rates at computation nodes so that the slowest sender finishes as fast as possible
Comm. w/o Barrier Synchronization	Prioritize if the application requires more information from this coflow; else lower priority.
Aggregation Tree	- Drop flows that are likely to miss the deadline [26]. - Set higher priorities for all the flows in a coflow that has the earliest deadline [16].

Table 3: Coflows, their objectives, and mechanisms to achieve them.

the network among multiple coflows [7, 23, 24]. We expect more diversity in the policies to attain an objective and to mitigate failures in doing so than that in the communication patterns or the objectives themselves.

6 Related Networking Abstractions

Control Plane Abstractions Software Defined Networking (SDN) [20] is gaining wide adoption in both academia and industry. The primary objective of SDN is to replace control plane mechanisms with abstractions that allow systematic decomposition of the existing protocols into composable modules for reuse and conceptual separation of concerns. Techniques developed in the context of SDN might be useful for optimizing coflow implementations.

Data Plane Abstractions Unlike the control plane, the data plane has long had abstractions at different layers. The notion of bits in the physical layer forms frames in the link layer, which in turn are combined into packets in the network layer. On top of that, there exists the abstraction of flows between two processes. Not only are these abstractions intellectually appealing, but they make it easier to express, solve, and optimize many of the practical problems. Coflows aim to do the same for cluster computing applications.

7 Conclusion

We studied the communication requirements of diverse cluster computing applications and distilled corresponding solutions to identify a networking abstraction, *coflow*, which can represent diverse communication requirements. We proposed an API to design cluster applications using the coflow abstraction, and we explored how the network might be able to better optimize its decisions using application-level semantics made available through the coflow API.

Acknowledgments

We thank Tathagata Das, Ali Ghodsi, Ganesh Ananthanarayanan, Aurojit Panda, Matei Zaharia, and David Zats for useful feedback. This research is supported in part by NSF CISE Expeditions award CCF-1139158, gifts from Amazon Web Services, Google, SAP, Blue Goji, Cisco, Cloudera, Ericsson, General Electric, Hewlett Packard, Huawei, Intel, MarkLogic, Microsoft, NetApp, Oracle, Quanta, Splunk, VMware, a Facebook Fellowship, and by DARPA (contract #FA8650-11-C-7136).

8 References

[1] Apache Giraph. `http://incubator.apache.org/giraph`.
[2] Apache Hadoop. `http://hadoop.apache.org`.
[3] Apache Hama. `http://hama.apache.org`.
[4] Apache HDFS. `http://hadoop.apache.org/hdfs/`.
[5] Apache Hive. `http://hadoop.apache.org/hive`.
[6] M. Al-Fares et al. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, 2010.
[7] H. Ballani et al. Towards predictable datacenter networks. In *SIGCOMM*, 2011.
[8] P. Bodik et al. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM*, 2012.
[9] R. Chaiken et al. SCOPE: Easy and efficient parallel processing of massive datasets. In *VLDB*, 2008.
[10] C. Chambers et al. FlumeJava: Easy, efficient data-parallel pipelines. In *PLDI*, pages 363–375, 2010.
[11] M. Chowdhury et al. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, 2011.
[12] T. Condie et al. MapReduce Online. In *NSDI*, 2010.
[13] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
[14] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.
[15] Z. Guo et al. Spotting code optimizations in data-parallel pipelines through PeriSCOPE. In *OSDI*, 2012.
[16] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. In *SIGCOMM*, 2012.
[17] M. Isard et al. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.
[18] Y. Low et al. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. In *PVLDB*, 2012.
[19] G. Malewicz et al. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.
[20] N. McKeown et al. OpenFlow: Enabling innovation in campus networks. *SIGCOMM CCR*, 38(2):69–74, 2008.
[21] C. Olston et al. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.
[22] R. Pike et al. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4), 2005.
[23] L. Popa et al. FairCloud: Sharing the network is cloud computing. In *SIGCOMM*, 2012.
[24] A. Shieh et al. Seawall: Performance Isolation for Cloud Datacenter Networks. In *HotCloud*, 2010.
[25] N. Tolia et al. An architecture for internet data transfer. In *NSDI*, 2006.
[26] C. Wilson et al. Better never than late: Meeting deadlines in datacenter networks. In *SIGCOMM*, 2011.
[27] Y. Yu et al. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.
[28] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.
[29] D. Zats et al. DeTail: Reducing the flow completion time tail in datacenter networks. In *SIGCOMM*, 2012.
[30] J. Zhang et al. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *NSDI*, 2012.