Performance optimization for distributed machine learning and graph processing at scale over virtualized infrastructure

Sun, Peng

2018

Sun, P. (2018). Performance optimization for distributed machine learning and graph processing at scale over virtualized infrastructure. Doctoral thesis, Nanyang Technological University, Singapore.

<http://hdl.handle.net/10356/73229>

<https://doi.org/10.32657/10356/73229>

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

**PERFORMANCE OPTIMIZATION FOR
DISTRIBUTED MACHINE LEARNING AND GRAPH
PROCESSING AT SCALE OVER VIRTUALIZED
INFRASTRUCTURE**

PENG SUN

**INTERDISCIPLINARY GRADUATE SCHOOL
ENERGY RESEARCH INSTITUTE @ NTU (ERI@N)**

2017

**PERFORMANCE OPTIMIZATION FOR
DISTRIBUTED MACHINE LEARNING AND GRAPH
PROCESSING AT SCALE OVER VIRTUALIZED
INFRASTRUCTURE**

PENG SUN

Interdisciplinary Graduate School
Energy Research Institute @ NTU (ERI@N)

A thesis submitted to the Nanyang Technological
University in partial fulfilment of the requirement for
the degree of
Doctor of Philosophy

2017

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research and has not been submitted for a higher degree to any other University or Institution.

31-Jul-2017

Peng Sun

.....
Date

.....
Student Name

Abstract

Nowadays, many real-world applications can be represented as machine learning and graph processing (MLGP) problems, and require sophisticated analysis on massive datasets. Various distributed computing systems have been proposed to run MLGP applications in a cluster. These systems usually manage the input data in a distributed file system (DFS), perform data-parallel computation on multiple machines, and exchange intermediate data via network. In this thesis, we focus on performance optimization of distributed MLGP over virtualized infrastructure.

First, we focus on improving the resource utilization of a cluster shared with multiple distributed MLGP workloads. Organizations are trending to use a cluster management system (CMS) to run multiple distributed MLGP applications in a single cluster. Existing CMSs can only allocate a static partition of the cluster to each application, leading to poor cluster utilization. To address this problem, we propose a new CMS named Dorm, which leverages virtualization techniques to partition a cluster, runs one application per partition, and can dynamically resize each partition at application runtime to achieve high cluster utilization and meet other performance constraints. Extensive performance evaluations have shown that Dorm could increase the cluster utilization by a factor of up to 2.32.

Second, we improve the metadata lookup performance for DFSs. Existing DFSs usually use distributed hash table (DHT) to manage their metadata servers. When performing a metadata operation, users should first use a lookup service to locate the desired metadata object. The lookup operation could lead to reduced metadata operation throughput and high latency. To address this problem, we design a new metadata lookup service called MetaFlow. MetaFlow leverages software-defined networking (SDN) techniques to transfer metadata lookup to the network layer, and generates appropriate flow tables for SDN-enabled switches by mapping the physical network topology to a logical B-tree. Extensive performance evaluations

have shown that MetaFlow could increase the system throughput by a factor of up to 6.5, and reduce the system latency by a factor of up to 5 for the metadata management, compared to DHT-based approaches.

Third, we reduce the communication overhead of distributed machine learning (ML) based on the Parameter Server (PS) framework. The PS framework has a group of worker nodes performing data-parallel computation, and has a group of server nodes maintaining globally shared parameters. Each worker node would continually pull parameters from server nodes and push updates to server nodes, resulting in high communication overhead. To address this problem, we design ParameterFlow, a communication layer for the PS framework with an update-centric communication (UCC) model and a dynamic value-bounded filter (DVF). UCC introduces a broadcast/push model to exchange data between worker nodes and server nodes. DVF could directly reduce network traffic and communication time by selectively dropping updates for network transmission. Experiments have shown that that PF could speed up popular distributed ML applications by a factor of up to 4.3, compared to the conventional PS framework.

Last, we enable high-performance large-scale graph processing in small clusters with limited memory. When processing big graphs, existing in-memory graph processing systems can easily exceed the cluster memory capacity. While out-of-core approaches could handle big graphs, they have poor performance due to high disk I/O overhead. We design a new distributed graph processing system named GraphH with three techniques: a gather-apply-broadcast computation model, an edge cache system and a hybrid communication mode. Experiments have shown that GraphH outperforms existing out-of-core systems by more than 100x, when processing big graphs in small clusters with limited memory.

The proposed approaches and obtained results can provide guidelines to improve large-scale distributed MLGP applications over virtualized infrastructure.

Acknowledgements

I would like to acknowledge the people who supported me in completing this work during my study at NTU.

First, I would like to express my thanks to my supervisor, Dr. Yonggang Wen, my co-supervisor Mr. Wong Yew Wah and my mentor Dr. Jianfei Cai. Under their supervision and training, I have developed skills in logical thinking, technical writing, communication skills and leadership, which will be invaluable to my future career.

Beyond my supervisors, I would like to thank Dr. Ta Nguyen Binh Duong, Dr. Xiaokui Xiao, Dr. Shengen Yan and Dr. Haiyong Xie, who have provided many insightful comments and contributed significantly to my research. In addition, I also would like to thank in advance the thesis examiners for their comments and suggestions to improve the quality of this thesis.

I would like to thank my friends and colleagues in IGS- ERI@N and SCSE. They have not only provided me with their valuable suggestions for my research, but also enriched my Ph.D. life at NTU with enjoyable experiences.

Finally, I cannot end without giving my special thanks to my parents and my wife, for their unselfish love and unconditioned supports.

Table of Contents

Abstract	i
Acknowledgements	iii
Table of Contents	v
Table Captions	xi
Figure Captions	xiii
Abbreviations	xxi
 Chapter 1 Introduction	1
1.1 Background.....	2
1.1.1 Single-Node Computing Systems are Insufficient	2
1.1.2 Overview of Distributed File Systems.....	3
1.1.3 Overview of Distributed Machine Learning.....	4
1.1.4 Overview of Distributed Graph Processing.....	5
1.1.5 Overview of Shared Cluster Management.....	6
1.2 Challenges	7
1.2.1 Low Resource Utilization of Shared Clusters	8
1.2.2 High Metadata Lookup Overhead of DFSs	8
1.2.3 High Communication Overhead of Distributed Machine Learning	9
1.2.4 Memory Overflow Problem of Distributed Graph Processing.....	9
1.3 Objectives	10
1.4 Organizations of the Chapters	10
References.....	12

Chapter 2 Literature Review.....	17
2.1 Cluster Management Systems	18
2.1.1 Static/Dynamic DCS-Level Resource Allocation.....	19
2.1.2 Static App-Level Resource Allocation	20
2.1.3 Dynamic Task-Level Resource Allocation	21
2.1.4 Discussion.....	22
2.2 Distributed Metadata Lookup Services	23
2.2.1 Hash-based Mapping	23
2.2.2 Distributed Hash Table	24
2.2.3 Discussion.....	24
2.3 Network Optimization for Distributed Machine Learning	25
2.4 Large-Scale Distributed Graph Processing.....	27
2.4.1 Graph Partitioning	27
2.4.2 Computation Engine	29
2.4.3 Discussion.....	32
2.5 Summary.....	33
References.....	33
 Chapter 3 Dorm: Dynamic Cluster Management for Distributed Machine Learning and Graph Processing.....	 41
3.1 Introduction	42
3.2 Dorm System Design.....	44
3.2.1 System Architecture.....	44
3.2.2 Dynamically-Partitioned Resource Allocation	47
3.3 Utilization-Fairness Optimizer	50
3.3.1 Objectives	50
3.3.2 Making Resource Allocation Decisions	52
3.3.3 Determining Newly Launched Applications	54
3.4 Implementation Details.....	55
3.5 Performance Evaluations.....	57

3.5.1	Experiments' Parameters and Configurations.....	58
3.5.2	Cluster Performance with Online Workloads	60
3.5.3	Cluster Performance with Offline Workloads.....	66
3.5.4	Application Speedup.....	69
3.6	Summary.....	69
	References.....	70

Chapter 4 MetaFlow: A Scalable Metadata Lookup Service 73

4.1	Introduction	74
4.2	Identify Bottlenecks in DHT-based Systems	76
4.2.1	Experiment Configurations.....	77
4.2.2	Throughput	77
4.2.3	Latency	80
4.3	MetaFlow System Design.....	81
4.3.1	Design Objective	82
4.3.2	System Architecture.....	82
4.3.3	MetaFlow Packets Forwarding.....	85
4.4	Flow Table Management	86
4.4.1	Flow Table Generation	87
4.4.2	Flow Table Maintenance.....	93
4.5	Performance Evaluations.....	96
4.5.1	Experiments' Parameters and Configurations.....	96
4.5.2	Throughput Results.....	98
4.5.3	Latency Results.....	100
4.5.4	Real-world Distributed File System Results	102
4.5.5	Real-world Application Results.....	104
4.6	Summary.....	105
	References.....	106

Chapter 5 ParameterFlow: Optimizing Network Performance for Large-

Scale Distributed Machine Learning	109
5.1 Introduction	110
5.2 Distributed ML on the PS Framework.....	112
5.2.1 ML: A Primer.....	112
5.2.2 Distributed ML over the PS framework	114
5.2.3 Synchronization Models	115
5.2.4 Communication Overhead Analysis using AlexNet	117
5.3 ParameterFlow System Design.....	117
5.3.1 System Overview.....	118
5.3.2 Update-Centric Communication Model	119
5.3.3 Dynamic Value-Bounded Filter	121
5.4 Convergence Analysis	124
5.5 Performance Evaluations	127
5.5.1 Experiments' Configuration.....	127
5.5.2 Reduced Network Traffic	128
5.5.3 Communication Time vs. Computation Time	129
5.5.4 Application Speedup.....	131
5.6 Summary.....	132
References.....	132
 Chapter 6 GraphH: Big Graph Analytics in Small Clusters.....	 135
6.1 Introduction	136
6.2 GraphH System Design	139
6.2.1 Notations.....	139
6.2.2 System Architecture.....	140
6.2.3 Spark-based Graph Pre-Processing	141
6.2.4 GAB-based Vertex-Centric Computation	145
6.3 System Optimizations.....	149
6.3.1 All-in-All Vertex Replication	149
6.3.2 Edge Cache	151

6.3.3	Hybrid Communication	154
6.4	Performance Evaluations	156
6.4.1	Effect of Compressed Edge Caching	157
6.4.2	PageRank Performance.....	158
6.4.3	SSSP Performance	160
6.5	Summary.....	161
	References.....	162
 Chapter 7 Summary and Future Work		165
7.1	Summary.....	166
7.1.1	Platform Optimizations.....	166
7.1.2	Framework Optimizations	168
7.2	Future Work	169
7.2.1	Resource-Efficient and QoS-Aware Cluster Management	169
7.2.2	Efficient Large-Scale Graph Processing on a Single Machine.....	170
7.2.3	High-Performance Time-Evolving Graph Analytics at Scale.....	170

Table Captions

Table 2.1	Existing approaches for reducing the communication overhead of large-scale distributed machine learning.	25
Table 3.1	Summary of notations used in Dorm.	49
Table 5.1	Summary of notations used in Parameter Flow.	113
Table 6.1	Graph datasets.	136
Table 6.2	Input data size (GB) for different graph processing systems. ..	145
Table 6.3	Compression ratio and processing throughput per CPU core. .	152
Table 6.4	Graph data size in tiles after compression.	154
Table 6.5	Average execution time per superstep to run PageRank and SSSP on EU-2015 using 9 servers with different edge cache modes. We collect memory usage of each server to show relationship between execution and memory footprint.	158

Figure Captions

Figure 1.1	Architecture of a typical distributed file system. It consists of three main components: clients, metadata nodes, and data nodes.	3
Figure 1.2	Architecture of the Parameter Server framework. Worker nodes perform data-parallel computation, and server nodes manages globally shared parameters.	4
Figure 1.3	Architecture of Pregel and PowerGraph. Pregel uses a hash function to assign a vertex to a server. PowerGraph allows a vertex to have multiple replicas.	5
Figure 1.4	VM vs. Container. Compared to VMs, containers do not run dedicated guest operation systems. They share the host operating system kernel and make use of guest operating system libraries. Thus, containers start much faster than VMs.	6
Figure 1.5	Main content and organization of the thesis.	11
Figure 2.1	Taxonomy of existing CMSs. Circles represent tasks, gray boxes are servers, and M denotes a distributed resource manager.	19
Figure 2.2	Existing distributed metadata management system architecture. Each node has two subsystems to process lookup requests and I/O requests separately. An I/O operation must wait for the completion of its associated lookup operation.	23
Figure 2.3	Graph partitioning strategies of Pregel+, PowerGraph, GraphD and Chaos. In (a) (c), each vertex is assigned to a server based on a hash function. In (b), the black cycle denotes a vertex replica. ..	28
Figure 3.1	Dorm's system architecture. Dorm consists of a central DormMaster to manage a set of DormSlaves. Utilization-fairness optimizer is a module in DormMaster to make resource allocation decisions with cluster-wide visibility.	44
Figure 3.2	An example to show how Dorm performs resource allocation. In this example, there are two running applications in the cluster.	

	When a user submits the third application, Dorm adjusts existing resource allocations to run the new application.	48
Figure 3.3	Dorm implementation details.	56
Figure 3.4	Distribution of application duration using single container in simulation.	59
Figure 3.5	Cluster resource utilization with online workloads in testbed (a, b, c, d) and simulation (e, f, g, h). The red line is the baseline system's resource utilization.	61
Figure 3.6	Fairness loss with online workloads. In this set of experiments, we adjust the value of θ_1 (the threshold of fairness loss) and use the same value of θ_2 (the threshold of resource adjustment overhead).	63
Figure 3.7	Resource adjustment overhead with online workloads. In this set of experiments, we adjust the value of θ_2 (the threshold of resource adjustment overhead) and use the same value of θ_1 (the threshold of fairness loss).	64
Figure 3.8	Cluster resource utilization with offline workloads in testbed (a, b, c, d) and simulation (e, f, g, h). The red line is the baseline system's resource utilization.	65
Figure 3.9	Fairness loss with offline workloads. In this set of experiments, we adjust the value of θ_1 (the threshold of fairness loss) and use the same value of θ_2 (the threshold of resource adjustment overhead).	67
Figure 3.10	Resource adjustment overhead with offline workloads. In this set of experiments, we adjust the value of θ_2 (the threshold of resource adjustment overhead) and use the same value of θ_1 (the threshold of fairness loss).	68
Figure 3.11	Application speedup ratio with online and offline workloads. ..	69
Figure 4.1	Existing distributed metadata management system architecture. Each node has two subsystems to process lookup requests and I/O	

	requests separately. An I/O operation must wait for the completion of its associated lookup operation.	75
Figure 4.2	Metadata operation throughput comparison between three overlay-based metadata management systems (Control Coordinator, Chord and One-Hop) in the testbed with four storage subsystems: MySQL, LevelDB (HDD), LevelDB (SSD), and Redis.	78
Figure 4.3	The lookup subsystem's CPU overhead in Chord and One-Hop based metadata management systems using four types of storage subsystems: MySQL (HDD), LevelDB (HDD), LevelDB (SSD), and Redis.	79
Figure 4.4	Metadata operation latency comparison between three overlay-based metadata management systems (Control Coordinator, Chord and One-Hop) in the testbed with four storage subsystems: MySQL, LevelDB (HDD), LevelDB (SSD), and Redis.	80
Figure 4.5	The lookup subsystem's latency overhead in Chord and One-Hop based metadata management systems using four types of storage subsystems: MySQL (HDD), LevelDB (HDD), LevelDB (SSD), and Redis.	81
Figure 4.6	A MetaFlow-based metadata management system architecture. There are two clusters in the system: the storage cluster hosts a set of storage servers to provide metadata service with a tree topology; and the application cluster manages a set of clients to query metadata objects. MetaFlow harnesses SDN to forward MetaFlow Requests to their associated storage servers using MetaDataIDs as the destination IP addresses.	83
Figure 4.7	A B-tree example. In this B-tree, there are 2 inner nodes and 6 leaf nodes. Each node can store 2 keys at most.	88
Figure 4.8	Mapping a three-tier tree network to a logical B-tree. The core layer switch is mapped to the B-tree's root node; the aggregation and edge layer switches are mapped to the inner nodes; and the	

	storage servers are mapped to the leaf nodes.	89
Figure 4.9	Mapping a fat tree network to a logical B-tree. In this network, each switch has four switch ports. Two aggregation layer switches, two edge layer switches, and four storage servers form a Pod in the fat tree network. To map the fat tree network to a B-tree, the four core layer switches are mapped to one B-tree's root node. The two aggregation layer switches in the same Pod are mapped to one inner node. Each edge layer switch and storage server is mapped to an inner node and a leaf node, respectively.	90
Figure 4.10	Using CIDR blocks to generate flow tables in a logical B-tree.	92
Figure 4.11	Splitting a B-tree node. In this example, when ServerA is full, the MetaFlow controller activates ServerC, uses 80.0.0.0 to split ServerA, and transfers the CIDR block 80.0.0.0/4 from ServerA to ServerC.	94
Figure 4.12	Throughput comparison in the simulation between the MetaFlow-based system and two DHT-based systems (Chord and One-Hop) with four lookup/storage throughput ratios: 100, 2, 1.5, and 1.	99
Figure 4.13	Throughput comparison between the MetaFlow-based system and two DHT-based systems (Chord and One-Hop) using the testbed with four types of storage subsystems: MySQL, LevelDB (HDD), LevelDB (SSD), and Redis.	100
Figure 4.14	Latency comparison in the simulation between the MetaFlow-based system and two DHT-based systems (Chord and One-Hop) with four lookup/storage latency ratios: 0.001, 0.5, 0.7, and 1.	101
Figure 4.15	Latency comparison between the MetaFlow-based system and two DHT-based systems (Chord and One-Hop) using the testbed with four storage subsystems: MySQL, LevelDB (HDD), LevelDB (SSD), and Redis.	102
Figure 4.16	Distributed file system's performance comparison between	

	MetaFlow and DHT-based approaches. The system contains 100 storage servers and 10 metadata servers. 50 clients are set up to generate background metadata workload in which 20% are get and 80% are put operations. We measure the time for a client to write 100 GB of files into the distributed file system with different file sizes: 64 KB, 256 KB, 16 MB, and 64 MB.	103
Figure 4.17	CDF distribution of input image data size.	104
Figure 5.1	System architecture of the Parameter Server framework.	114
Figure 5.2	The BSP and SSP synchronization models. With BSP, there is a synchronization barrier at the end of each iteration. With SSP, worker nodes can use cached parameters to perform computation. In this example, the staleness threshold is 2. Thus, to perform computation of clock 3, a worker node needs to see effects of updates with timestamp 0 from all other worker node.	115
Figure 5.3	System architecture of a PF-enabled PS framework.	118
Figure 5.4	DVF allows worker nodes and server nodes to selectively drop updates with a given threshold during the push or broadcast operations.	122
Figure 5.5	We use a testbed with a tier tree topology to evaluate PF.	127
Figure 5.6	Percentage of reduced network traffic when using PF to train LR, MF and AlexNet in the testbed with two thresholds.	129
Figure 5.7	Communication and computation time used to train LR, MF and AlexNet with various staleness threshold (threshold of 0 denotes BSP).	130
Figure 5.8	PF can speed up distributed ML with BSP and SSP. In the experiments, we use δ_{up} in PF-BSP, and use δ_{low} in PF-SSP-16. The baseline system uses BSP.	131
Figure 6.1	Memory requirements and execution time for running PageRank on UK-2007 with various distributed graph processing systems. The testbed has 9 servers, and each server contains 12x2.0GHz	

	cores (two Intel Xeon E5-2620 CPUs), 128GB memory, 4x4TB HDDs and 10Gbps Ethernet. We use Giraph-1.1, GraphX-2.0, PowerGraph-2.2 and the newest version of Pregel+, GraphD and Chaos.	137
Figure 6.2	GraphH system architecture.	141
Figure 6.3	Graph data pre-processing overview. SPE splits the raw graph into tiles, each of which contains a similar number of edges in CSR format. Edges appear in the same tile as their target vertex.	142
Figure 6.4	GAS vs. GAB. The GAS model requires three computation operations (local sum in the gather phase, update vertex in the apply phase, and active neighbors in the scatter phase) and two communication operations to update a vertex. The GAB model needs one computation operation (gather and apply), and one communication operation (broadcast updated vertex values). .	149
Figure 6.5	GraphH memory usage analytics. (a) shows the expected memory usage per server, when running PageRank with the All-in-All policy and On-Demand policy. (b) shows the memory usage of GraphH (All-in-All policy without edge cache) per server in a 9-node cluster to run PageRank and SSSP. Each server has 128GB memory.	151
Figure 6.6	Execution time and cache hit ratio comparison with different cache modes. We run PageRank on EU-2015 in this set of experiments.	153
Figure 6.7	Network traffic of PageRank on UK-2007 in a 9-node cluster. In (a), we show the vertex update ratio (i.e., the percentage of vertices which update their values) of a superstep during the computation. In (b), we compare the network traffic size of a superstep between sparse and dense communication mode. In (c), we show the network traffic size of a superstep when using the hybrid communication mode. In (d), we show the execution time of a	

	superstep with hybrid communication mode.	155
Figure 6.8	In-degree and our-degree distribution of Twitter-2010, UK-2007, UK-2014 and EU-2015. These four graphs follow skewed power-law distribution.	157
Figure 6.9	Average execution time per superstep to run PageRank on four graph datasets (Twitter-2010, UK-2007, UK-2014 and EU-2015) using six distributed graph processing systems (GraphH, Pregel+, PowerGraph, PowerLyra, GraphD and Chaos). In (c) (d), we only show the performance of GraphH, GraphD and Chaos, since Pregel+, PowerGraph, PowerLyra cannot run PageRank on UK-2014 and EU-2015.	159
Figure 6.10	Average execution time per superstep to run SSSP on four graph datasets (Twitter-2010, UK-2007, UK-2014 and EU-2015) using six distributed graph processing systems (GraphH, Pregel+, PowerGraph, PowerLyra, GraphD and Chaos). In (c) (d), we only show the performance of GraphH, GraphD and Chaos, since Pregel+, PowerGraph, PowerLyra cannot run SSSP on UK-2014 and EU-2015.	160

Abbreviations

BAP	Bounded Asynchronous Parallel
BSP	Bulk Synchronous Parallel
CC	Connected Components
CMS	Cluster Management System
CNN	Convolutional Neural Networks
CSR	Compressed Sparse Row
DCS	Distributed Computing System
DFS	Distributed File System
DHT	Distributed Hash Table
DRF	Dominant Resource Fairness
DVF	Dynamic Value-Bounded Filter
GAB	Gather-Apply-Broadcast
GAS	Gather-Apply-Scatter
LR	Logistic Regression
MF	Matrix Factorization
ML	Machine Learning
MLGP	Machine Learning and Graph Processing
MPI	Message Passing Interface
P2P	Peer to Peer
PS	Parameter Server
PF	ParameterFlow
QoS	Quality of Service
SDN	Software-Defined Networking
SSP	Stale Synchronous Parallel
SSSP	Single Source Shortest Path
UCC	Update-Centric Communication
VAP	Value-Bounded Synchronous Parallel
VM	Virtual Machine

Chapter 1

Introduction

This chapter first presents the background of large-scale distributed machine learning and graph processing (MLGP). We then identify the limitations and shortcomings of existing approaches to distributed MLGP, and illustrate the objectives of our work. Finally, we point out the key contributions and the organization of the thesis.

1.1 Background

1.1.1 Single-Node Computing Systems are Insufficient

In the era of “Big Data”, many real-world problems, such as social network analytics, collaborative recommendation and natural language processing, can be represented as machine learning and graph processing (MLGP) problems. It has attracted considerable interest in both academia and industry to process massive datasets and extract valuable knowledge from them.

Many systems have been proposed to solve MLGP problems on a single server. For example, Theano [1] and scikit-learn [2] are two popular machine learning systems. Ligra [3] and GraphMat [4] are widely used to process graph datasets. These systems usually manage the input datasets on local disks, and use multiple CPUs or GPUs to execute user-defined functions for a specific MLGP application.

However, aforementioned single-node computing systems cannot handle big data, due to the limitations in storage and computation resources. Modern industrial MLGP applications usually need to process terabytes of data. For example, the Yahoo News Feed dataset consists of ~110B lines (1.5TB after compression) of user-news interaction data [5]; ImageNet contains ~14 million labeled images [6]; ClueWeb12 has ~733 million web pages [7]. When using single-node systems to process big data, they may easily crash due to the out-of-memory issues, or take unacceptable running time (e.g., several months).

Therefore, various distributed computing systems (DCSs) have been proposed to run MLGP applications on multiple servers in a cluster. These systems usually store the input datasets in a distributed file system (DFS), which can be accessed by all computation servers via network. Each server is in charge of processing a part of the input datasets, and exchanges intermediate results with other nodes.

Figure 1.1 Architecture of a typical distributed file system. It consists of three main components: clients, metadata nodes, and data nodes.

1.1.2 Overview of Distributed File Systems

Metadata is “data about data”. It summarizes the basic information regarding files and directories in file systems. Normally, a metadata object is a key-value pair, where the key denotes the file name, and the value consists of a set of attributes (e.g., file size, permission, access time, disk block) of the file or directory. In file systems, before users could perform file-related operations, such as open, read, write, they have to acquire the files’ metadata first.

DFSs like HDFS [8], GFS [9], Lustre [10] and Ceph [11] usually separate the management of metadata from storage servers. Such separation could make it easier to scale the storage capacity of the file system, since new storage servers can be added to the cluster when needed. As shown in Figure 1.1, a typical DFS contains three components: clients, each of which exposes a file system interface to a host or process; a set of data nodes, which collectively store all data; and a set of metadata nodes to manage all metadata. Clients should interact with metadata servers first to fetch files’ addresses and other attributes, after that they could perform operations on the desired files. Previous studies have shown that more than 80% of file operations need to interact with metadata servers [12], [13].

Figure 1.2 Architecture of the Parameter Server framework. Worker nodes perform data-parallel computation, and server nodes manages globally shared parameters.

1.1.3 Overview of Distributed Machine Learning

Machine learning (ML) builds models from training datasets, and use them to make predictions on new data. Typically, an ML model consists of a number of parameters. To minimize the prediction error, an ML application usually uses an iterative-convergent algorithm, such as stochastic gradient descent (SGD), to train a given ML model.

To handle massive training datasets, various distributed ML systems have been proposed based on the Parameter Server (PS) framework [14], such as Petuum [15], MxNet [16], Project Adam [17], SINGA [18], and TensorFlow [19]. Figure 1.2 shows the system architecture of the PS framework. The PS framework can scale to large cluster deployments by having a set of worker nodes performing data-parallel computation, and having a set of server nodes maintaining globally shared parameters. When training an ML model, worker nodes continuously pull latest parameters from server nodes, perform computation on partitions of the training dataset, and push generated updates to server nodes. The server nodes would aggregate all received updates, and use them to return a new version of parameters. This is done iteratively to bring each parameter closer to the optimal value, which could minimize the prediction error.

Figure 1.3 Architecture of Pregel and PowerGraph. Pregel uses a hash function to assign a vertex to a server. PowerGraph allows a vertex to have multiple replicas.

1.1.4 Overview of Distributed Graph Processing

Many distributed graph processing systems have been proposed to tackle graph analytics in memory. They usually follow the “think as a vertex” philosophy, and abstract graph processing as vertex-centric programs. Pregel and GAS (Gather-Apply-Scatter) are two widely used graph computation models. Most the existing distributed graph processing systems are designed based on these two models.

- **Pregel-like Systems.** Pregel [20], Giraph [21], GPS [22] and HuSky [23] adopt the Pregel computation model: they assign the input graph’s vertices to multiple servers, and provide interaction between them with message passing along out-edges, as shown in Figure 1.3 (a).
- **GAS-based Systems.** PowerGraph [24], PowerLyra [25], GraphX [26] and LFGGraph [27] use the GAS model: they split a vertex into multiple replicas, and parallelize the computation for a single vertex on different servers, as shown in Figure 1.3 (b).

Recent benchmarking results have shown that aforementioned systems can offer better performance than general-purpose systems like Hadoop and Spark [28]. It should be noted that these systems need to manage the entire input graph and all intermediate results into memory during the computation.

Figure 1.4 VM vs. Container. Compared to VMs, containers do not run dedicated guest operation systems. They share the host operating system kernel and make use of guest operating system libraries. Thus, containers start much faster than VMs.

1.1.5 Overview of Shared Cluster Management

Driven by the emerging MLGP applications, researchers have been developing a diverse array of distributed computing systems (DCSs). Existing studies have shown that no system will be optimal for all MLGP applications [29]. Therefore, organizations are trending to run multiple DCSs systems in the same cluster, so that users can pick the most suitable system for each specific MLGP application. Existing approaches to sharing a cluster with multiple MLGP workloads can be categorized into two resource allocation models.

- **Static DCS-level model.** This solution statically partitions a cluster, and runs one DCS per partition. For example, we can use OpenStack [30] to create a number of Virtual Machines (VMs) for MxNet, and execute all MxNet-based applications in this virtual cluster.
- **Static app-level model.** This solution can statically allocate a number of

resources to applications, and let each application reserve acquired resources until completion. For example, we can use Yarn [31] or Mesos [29] to create a set of containers to run a TensorFlow-based application.

Two types of virtualization techniques are widely used for cluster management: hardware virtualization and operating-system-level virtualization. In Figure 1.4, we summarize the difference between these two types of virtualization techniques. More specifically, hardware virtualization is the virtualization of computers as complete hardware platforms. It installs a hypervisor to create an abstraction layer between the software and the underlying hardware. Once a hypervisor is in place, it could create a simulated environment, a VM, for its guest software. As a comparison, operating-system-level virtualization (which is also known as container-based virtualization or containerization) deploy and run applications without launching an entire VM for each application. With containerization, multiple isolated instances, called containers, are run on a single host and access a single operating system kernel. Because containers share the same operating system kernel as the host, containers could be more efficient than VMs, which require separate operating systems.

1.2 Challenges

Performance is the key metric for distributed MLGP. This section motivates the importance of our work by showing the problems and challenges on four aspects. First, when running multiple distributed MLGP applications in a single cluster, the low cluster utilization problem could result in poor application performance. Second, in large-scale DFSs, existing metadata lookup service can significantly degrade the throughput and latency of the metadata management system. Third, the high communication overhead could considerably reduce the performance of distributed ML. Four, it is hard for existing distributed graph processing systems to handle big graphs in memory due to the serious memory overflow problem.

1.2.1 Low Resource Utilization of Shared Clusters

We could deploy and run multiple DCSs in a single cluster with existing cluster sharing solutions. However, these solutions, including static DCS-level model and static app-level model, may lead to quite low cluster utilization, since busy DCSs or applications cannot dynamically scale out to use idle cluster resources.

It should be noted that the **dynamic task-level resource allocation model**, which is used by Hadoop, cannot work for distributed MLGP workloads. In a dedicated Hadoop cluster, all computation nodes are subdivided into “slots”. Each Hadoop MapReduce application is composed of a set of independent short tasks, which are matched to slots. Short tasks allow the cluster to achieve high utilization, as applications can rapidly scale when new slots become available. When running a Hadoop MapReduce application in a shared cluster managed by Yarn or Mesos, it can still maximize the cluster utilization by continually acquiring resources to execute its tasks. However, most of the distributed MLGP systems, such as Pregel, PowerGraph and MxNet, can start an application only after it has acquired a fixed quantity of resources, and cannot dynamically scale out during runtime.

1.2.2 High Metadata Lookup Overhead of DFSs

As shown in Section 1.1.2, modern DFSs, such as GFS and Ceph, deploy a cluster to share the metadata workload. In these systems, a very large metadata table is partitioned into smaller parts, which are located on separate metadata servers. Existing approaches, e.g., [11], [32], [33], focused on building overlay-based metadata management systems. These systems provide two main operations: lookup and I/O. In particular, the lookup operation aims to locate the desired metadata, and the I/O operation retrieves the metadata itself from the storage server using the address returned from the lookup operation. In this way, the metadata workload could be distributed across a cluster.

However, the throughput and latency of these overlay-based systems could be significantly degraded due to the bottleneck created by a large number of lookup operations. Such operations compete for CPU cycles with I/O operations, which might lead to reduced system throughput. On the other hand, it could take a long time to locate a metadata object in overlay-based systems, which might increase the system latency. Hence, it is important to reduce the lookup overhead of distributed metadata management for large-scale DFSs.

1.2.3 High Communication Overhead of Distributed Machine Learning

A major challenge of large-scale distributed ML is the high communication overhead. Modern ML applications are trending to learn complex ML models with hundreds of millions of parameters. For example, VGG-19 [34] contains 175M parameters, and the multi-class logistic regression (MLR) model trained from Wikipedia has billions of parameters [35]. When using the PS framework, each worker node should push all updates to the server nodes and pull a huge number of parameters frequently, generating a large amount of network traffic. Due to the limited bandwidth of commodity network, each worker node spends a significant portion of time on network communication. Our experiments have shown that the communication time could be much more than the computation time in many cases. Therefore, it is important to reduce the communication overhead and improve network performance for large-scale distributed ML.

1.2.4 Memory Overflow Problem of Distributed Graph Processing

As shown in Section 1.1.4, many distributed graph processing systems need to load the entire graph and manage all intermediate results in the main memory during the computation. This strategy is appropriate when processing generic graphs with a few billion edges. Unfortunately, it is common for real-world applications to analyze big graphs with billions of vertices and hundreds of

billions of edges. In this case, the input graph and intermediate data can easily exceed the cluster memory limit, leading to significant performance degradation or even program crashes. We evaluated the memory requirement for running PageRank on the UK-2007 dataset (which has 124M vertices and 5.5B edges) with 5 distributed graph processing systems in a 9-node cluster. The experiment results showed that Giraph, GraphX, PowerGraph, PowerLyra and Pregel+ need 795GB, 685GB, 357GB, 511GB and 281GB memory, respectively. To process a big graph with 1 billion vertices, these in-memory graph processing systems even require a large cluster with at least 5TB memory. Hence, it is crucial to solve the memory overflow problem of current distributed graph processing systems, and enable big graph analytics in a small cluster with limited memory.

1.3 Objectives

In this thesis, we focus on performance optimization for large-scale distributed MLGP over virtualized infrastructure. The main objectives of this work are summarized as follows:

- Design an efficient cluster management system called Dorm to improve the cluster utilization using virtualization techniques.
- Design a scalable metadata lookup service called MetaFlow for large-scale DFSs to improve the metadata lookup performance using software-defined networking (SDN) techniques.
- Design a novel communication layer called ParameterFlow to reduce the communication overhead of large-scale distributed ML.
- Design a new distributed graph processing system called GraphH to enable big graph analytics in a small cluster with limited memory.

1.4 Organizations of the Chapters

Figure 1.4 shows the main content and organization of the thesis. We can divide

Figure 1.5 Main content and organization of the thesis.

the main content into two parts. The first half of the thesis (Chapter 3 and 4) considers the platform optimizations for distributed MLGP. The second half (Chapter 5 and 6) considers the framework optimizations for distributed MLGP.

The rest of this thesis is organized as follows

- **Chapter 2** reviews some related work on performance optimization of large-scale distributed MLGP, including cluster management, distributed metadata lookup services, network optimizations for large-scale distributed ML and memory optimizations for big graph analytics.

- **Chapter 3** aims to improve the resource utilization of a cluster shared with multiple distributed MLGP workloads. In this chapter, we propose a new cluster management system called Dorm using virtualization techniques.
- **Chapter 4** aims to reduce the metadata lookup overhead for large-scale DFSs. In this chapter, we propose MetaFlow, an SDN-based lookup service.
- **Chapter 5** aims to improve the network performance for existing distributed ML systems. In this chapter, we propose a new communication layer called ParameterFlow to reduce the communication overhead of distributed ML.
- **Chapter 6** aims to address the memory overflow problem of existing distributed in-memory graph processing systems. In this chapter, we propose GraphH to enable big graph analytics in small clusters with limited memory.
- **Chapter 7** summarizes the thesis and discusses the further work.

References

- [1] Bergstra, J., Breuleux, O., Lamblin, P., Pascanu, R., Delalleau, O., Desjardins, G., Goodfellow, I., Bergeron, A., Bengio, Y. and Kaelbling, P., 2011. Theano: Deep learning on gpus with python.
- [2] Pedregosa, Fabian, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel et al. "Scikit-learn: Machine learning in Python." *Journal of Machine Learning Research* 12, no. Oct (2011): 2825-2830.
- [3] Shun, Julian, and Guy E. Blelloch. "Ligra: a lightweight graph processing framework for shared memory." In *ACM Sigplan Notices*, vol. 48, no. 8, pp. 135-146. ACM, 2013.
- [4] Sundaram, Narayanan, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. "Graphmat: High performance graph analytics made productive." *Proceedings of the VLDB Endowment* 8, no. 11 (2015): 1214-1225.
- [5] <https://webscope.sandbox.yahoo.com/catalog.php?datatype=r>

- [6] Deng, Jia, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. "Imagenet: A large-scale hierarchical image database." In Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on, pp. 248-255. IEEE, 2009.
- [7] <http://lemurproject.org/clueweb12/>
- [8] Shvachko, Konstantin, Hairong Kuang, Sanjay Radia, and Robert Chansler. "The hadoop distributed file system." In Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on, pp. 1-10. IEEE, 2010.
- [9] Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google file system." In ACM SIGOPS operating systems review, vol. 37, no. 5, pp. 29-43. ACM, 2003.
- [10] Schwan, Philip. "Lustre: Building a file system for 1000-node clusters." In Proceedings of the 2003 Linux symposium, vol. 2003, pp. 380-386. 2003.
- [11] Weil, Sage A., Scott A. Brandt, Ethan L. Miller, Darrell DE Long, and Carlos Maltzahn. "Ceph: A scalable, high-performance distributed file system." In Proceedings of the 7th symposium on Operating systems design and implementation, pp. 307-320. USENIX Association, 2006.
- [12] Carns, Philip, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. "Small-file access in parallel file systems." In Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, pp. 1-11. IEEE, 2009.
- [13] Roselli, Drew S., Jacob R. Lorch, and Thomas E. Anderson. "A Comparison of File System Workloads." In USENIX annual technical conference, general track, pp. 41-54. 2000.
- [14] Li, Mu, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. "Scaling Distributed Machine Learning with the Parameter Server." In OSDI, vol. 1, no. 10.4, p. 3. 2014.

- [15] Xing, Eric P., Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. "Petuum: A new platform for distributed machine learning on big data." *IEEE Transactions on Big Data* 1, no. 2 (2015): 49-67.
- [16] Chen, Tianqi, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems." *arXiv preprint arXiv:1512.01274* (2015).
- [17] Chilimbi, Trishul M., Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. "Project Adam: Building an Efficient and Scalable Deep Learning Training System." In *OSDI*, vol. 14, pp. 571-582. 2014.
- [18] Ooi, Beng Chin, Kian-Lee Tan, Sheng Wang, Wei Wang, Qingchao Cai, Gang Chen, Jinyang Gao et al. "SINGA: A distributed deep learning platform." In *Proceedings of the 23rd ACM international conference on Multimedia*, pp. 685-688. ACM, 2015.
- [19] Abadi, Martín, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems." *arXiv preprint arXiv:1603.04467*(2016).
- [20] Malewicz, Grzegorz, Matthew H. Austern, Aart JC Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: a system for large-scale graph processing." In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135-146. ACM, 2010.
- [21] Avery, Ching. "Giraph: Large-scale graph processing infrastructure on hadoop." *Proceedings of the Hadoop Summit*. Santa Clara 11, no. 3 (2011): 5-9.
- [22] Salihoglu, Semih, and Jennifer Widom. "GPS: A graph processing system." In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, p. 22. ACM, 2013.

- [23] Yang, Fan, Jinfeng Li, and James Cheng. "Husky: Towards a more efficient and expressive distributed computing framework." *Proceedings of the VLDB Endowment* 9, no. 5 (2016): 420-431.
- [24] Gonzalez, Joseph E., Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs." In *OSDI*, vol. 12, no. 1, p. 2. 2012.
- [25] Chen, Rong, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. "Powerlyra: Differentiated graph computation and partitioning on skewed graphs." In *Proceedings of the Tenth Eurosys*, p. 1. ACM, 2015.
- [26] Xin, Reynold S., Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. "GraphX: A resilient distributed graph system on spark." In *First International Workshop on Graph Data Management Experiences and Systems*, p. 2. ACM, 2013.
- [27] Hoque, Imranul, and Indranil Gupta. "LFGraph: Simple and fast distributed graph analytics." In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, p. 9. ACM, 2013.
- [28] Capotă, Mihai, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, and Peter Boncz. "Graphalytics: A big data benchmark for graph-processing platforms." In *Proceedings of the GRADES'15*, p. 7. ACM, 2015.
- [29] Hindman, Benjamin, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." In *NSDI*, vol. 11, no. 2011, pp. 22-22. 2011.
- [30] Sefraoui, Omar, Mohammed Aissaoui, and Mohsine Eleuldj. "OpenStack: toward an open-source solution for cloud computing." *International Journal of Computer Applications* 55, no. 3 (2012).
- [31] Vavilapalli, Vinod Kumar, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves et al. "Apache hadoop yarn: Yet another resource negotiator." In *Proceedings of the 4th*

- annual Symposium on Cloud Computing, p. 5. ACM, 2013.
- [32] Aguilera, Marcos K., Wojciech Golab, and Mehul A. Shah. "A practical scalable distributed b-tree." *Proceedings of the VLDB Endowment* 1, no. 1 (2008): 598-609.
- [33] Li, Weijia, Wei Xue, Jiwu Shu, and Weimin Zheng. "Dynamic hashing: Adaptive metadata management for petabyte-scale file systems." In *23rd IEEE/14th NASA Goddard Conference on Mass Storage System and Technologies*, pp. 93-98. 2006.
- [34] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv:1409.1556*(2014).
- [35] Xie, Pengtao, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric P. Xing. "Lighter-Communication Distributed Machine Learning via Sufficient Factor Broadcasting." In *UAI*. 2016.

Chapter 2

Literature Review

In this chapter, we provide a literature review of performance optimizations for distributed MLGP. We first review existing cluster management systems used for sharing a single cluster with multiple distributed MLGP workloads. Next, we review current metadata lookup services and strategies used in large-scale DFSs. Then, we review existing approaches for reducing the communication overhead of large-scale distributed ML. Last, we review recently proposed distributed graph processing systems.

2.1 Cluster Management Systems

Cluster management systems (CMSs) are designed to run multiple distributed computing systems (DCSs) in a single cluster, so that users can select the best one for each specific application. For example, in a shared cluster with Hadoop and MPI, we can use Hadoop to execute data-intensive applications, and choose MPI to run CPU-intensive applications. Existing CMSs can be classified into six categories based on their resource management mechanisms, as shown in Figure 2.1. These CMSs could statically or dynamically allocate cluster resources at three levels: DCS, application and task. Some key concepts are defined as follows:

- **DCS.** DCS means a software system that can use multiple networked servers to run specific applications in a cluster. For example, Hadoop and Spark are two widely used general purpose DCSs. MxNet and TensorFlow are designed for large-scale distributed ML. Pregel and PowerGraph can process graph datasets in parallel. DCSs like Hadoop and Spark can manage and execute multiple applications in a cluster. Most of the DCSs designed for distributed MLGP applications can only execute one application at a time. For example, given a MxNet cluster, we can only start a new machine learning application after the completion of the running one.
- **Application.** Users can use the given APIs of a DCS to design an application for a specific purpose, and execute it in the corresponding DCS cluster. For example, we can use Pregel APIs to implement a PageRank application, and execute it in a dedicated Pregel cluster.
- **Task.** Each application is composed of a set of small tasks. Specifically, when running an application on a DCS, the DCS usually consists of a scheduler to coordinate the execution of this application's tasks. In this work, we assume that each application has its own task scheduler for simplicity.
- **Resource allocation** refers to determining the amount of resources offered to each application, and selecting specific resources from servers to satisfy user-supplied placement preferences or other constraints.

Figure 2.1 Taxonomy of existing CMSs. Circles represent tasks, gray boxes are servers, and M denotes a distributed resource manager.

2.1.1 Static/Dynamic DCS-Level Resource Allocation

Infrastructure-as-a-Service (IaaS) CMSs can share a cluster at the level of DCSs statically or dynamically, such as OpenStack [1] and CloudStack [2]. More specifically, this approach uses a centralized cluster resource manager to provide virtualized resources to each DCS. Users should submit their applications to corresponding DCSs, and let DCSs execute received applications on allocated resources. For example, we can use OpenStack to create a set of Virtual Machines (VMs), set up Hadoop on them, and run all Hadoop MapReduce applications in this virtual cluster. In addition, OpenStack could auto-scale the virtual Hadoop cluster to guarantee application performance [3]. IaaS CMSs require that DCSs have the ability to manage and schedule multiple applications. Otherwise, users should manually allocate resources of a virtual cluster to submitted applications.

2.1.2 Static App-Level Resource Allocation

Monolithic, two-level, shared-state, fully-distributed and hybrid CMSs support cluster sharing at the level of applications and tasks. In this section, we focus on static app-level cluster sharing: CMSs allocates static resources to applications, and let each application reserve acquired resources to run its tasks.

Monolithic CMSs use a central resource manager to perform resource allocation for all applications, such as Yarn [4], Quasar [5], Gemini [6], Paragon [7], Fuxi [8], Swarm [9], Borg [10] and Kubernetes [11]. With monolithic CMSs, each application sends its resource request to the central resource manager, and lets it select resources from multiple servers to satisfy user-specified constraints. Yarn employs FIFO, capacity and fair schedulers to provide multi-tenant support with capacity guarantees, elastic sharing and fairness. Gemini leverages a regression model to maximize the cluster utilization given user-specified fairness constraints. Quasar and Paragon use machine learning techniques to predict the right amount of resources for each application to meet its performance constraint. Fuxi designs an incremental resource management protocol to supports multi-dimensional resource allocation. Swarm, Borg and Kubernetes can share a cluster with both long-running services and DCS-based applications. For example, Google uses Borg to handle end-user-facing products (e.g., Gmail), infrastructure services (e.g., BigTable) and MapReduce applications in the same cluster.

Two-level CMSs use a central cluster resource manager and multiple application-specific schedulers to jointly perform cluster resource allocation, such as Mesos [12]. More specifically, Mesos uses a central node to allocates cluster resources to applications in the form of resource offers and achieves dominant resource fairness (DRF) [13]. Applications accept qualified resource offers, and deny those cannot satisfy user-defined constraints (e.g., data locality). The denied resource offers from an application may be accepted by other applications.

Shared-state CMSs allow concurrent applications to compete for resources using the global cluster state, and use lock-free optimistic concurrency control to solve resource conflicts, such as Omega [14], Apollo [15] and Tarcil [16]. This approach could offer high resource allocation quality with low latency, since each application could use its own logic to occupy any resources in the cluster with cluster-wide visibility. However, it is hard for shared-state CMS to achieve strict fairness guarantees due to the lack of centralized resource management.

Fully-distributed CMSs use multiple, independent cluster resource managers to perform resource allocation with local, partial and even stale cluster state, such as Sparrow [17]. More specifically, Sparrow makes each server have a resource manager to serve applications' resource requests. If a server receives more resource requests than it can offer right now, it queues new requests until enough resources are released. Sparrow could offer millisecond-level scheduling latency, and works best with low-latency workloads. However, fully-distributed CMSs may have sub-optimal resource allocation quality and high fairness loss, due to the lack of centralized resource management.

Hybrid CMSs combine distributed resource managers with a central cluster manager, such as Hawk [18] and Mercury [19]. Applications can obtain strong execution guarantees from the central manager, or trade strict guarantees for millisecond scheduling latency from distributed managers.

2.1.3 Dynamic Task-Level Resource Allocation

Monolithic, two-level, shared-state, fully-distributed and hybrid CMSs can also share cluster resources dynamically at the level of tasks. In this mode, each application immediately launches tasks on acquired resources, releases them as soon as the task finishes, and petitions for new resources to run pending tasks. In this way, applications can easily scale out to use idle cluster resources to execute

their tasks. Compared to static app-level resource allocation, dynamic task-level resource allocation could promise higher cluster utilization, allocation quality (e.g., data locality), and stronger fairness guarantee at the cost of higher sharing overhead, since tasks cannot start until receiving enough resources. However, only a small set of DCSs support this resource allocation mode, since it represents a lot of engineering challenges to make applications release occupied resources during runtime.

2.1.4 Discussion

In this work, we focus on sharing a cluster with distributed MLGP workloads. However, existing CMSs cannot efficiently handle distributed MLGP systems and applications, and could have poor cluster utilization problem.

- IaaS CMSs cannot efficiently handle distributed MLGP workloads at the level of DCSs, since popular distributed MLGP systems (e.g., PowerGraph, MxNet, TensorFlow) do not have built-in schedulers to manage multiple applications.
- When sharing cluster resources statically at the level of applications, existing CMSs can only allocate a static partition of the cluster to each application. Therefore, distributed MLGP applications cannot dynamically scale up/down or scale out/in to achieve high resource utilization.
- Popular distributed MLGP systems cannot support dynamic task-level cluster sharing due to engineering challenges and performance considerations.

In practice, existing CMSs could only allocate cluster resources to distributed MLGP workloads statically at the level of applications, such as TensorFlow-on-Mesos and MxNet-on-Yarn. In this case, when submitting a new distributed MLGP application, users must manually specify its resource demands, including the number of computation nodes, and the number of CPUs, GPUs and RAM per computation node. The application can be started only after having all required resources, and cannot use idle resources, resulting in poor cluster utilization.

Figure 2.2 Existing distributed metadata management system architecture. Each node has two subsystems to process lookup requests and I/O requests separately. An I/O operation must wait for the completion of its associated lookup operation.

2.2 Distributed Metadata Lookup Services

In large-scale DFSs with distributed metadata management systems, a metadata operation consists of two sequential operations: lookup and I/O. As shown in Figure 2.2, an I/O operation must wait for the completion of its associated lookup operation, since it must know the destination to send the I/O request. Generally, each node contains two subsystems to process lookup requests and I/O requests separately. The lookup subsystem maintains some storage information to locate metadata objects. The storage subsystem, which is usually a high-performance in-memory storage system, deals with the I/O operations on metadata objects. Lookup services for metadata management, which map a metadata object to its location, i.e., a metadata storage server, have been receiving much attention. In this section, we review the existing approaches of distributed metadata lookup.

2.2.1 Hash-based Mapping

Hash-based mapping [20] is one approach to locating metadata objects. This approach hashes a file name to an integer k , and assigns its metadata to a server

according to the remainder value when dividing k by the number of metadata servers. There is no lookup overhead on storage servers with this approach, since such lookup is done on the client side using the hash function. However, the hash-based mapping might not be practical, since all metadata objects must be re-allocated if a metadata server joins or leaves the system.

2.2.2 Distributed Hash Table

Existing large-scale DFSs are focusing on using the distributed hash table (DHT) for metadata management. In these systems, a lookup request is processed by several metadata servers, each of which maintains partial storage information for other servers. Two DHT-based approaches are widely used:

- **Chord** [21]. In an N -node system, each Chord node stores $O(\log N)$ other nodes' storage information. On average, each lookup request needs to interact with $O(\log N)$ nodes to locate a metadata object. Arpeggio [22], a peer-to-peer file-sharing network, uses Chord to support distributed metadata lookup.
- **One-Hop** [23]. One-Hop allows each node to store all other nodes' storage information. Therefore, any lookup requests will be processed by only one node. As shown in [24], One-Hop could achieve high throughput and low latency for metadata operations. Some emerging distributed storage systems, such as Amazon Dynamo [25], Apache Cassandra [26], are using One-Hop for general key-value data including metadata management.

2.2.3 Discussion

DHT-based metadata management systems need to process a large number of lookup operations. These lookup operations would compete for CPU cycles with in-memory I/O operations, leading to reduced throughput of metadata operations. Moreover, a lookup operation may take a long time to locate a metadata object using DHT, which might increase the latency of metadata operations.

Table 2.1 Existing approaches for reducing the communication overhead of large-scale distributed machine learning.

Reduced Communication Frequency	DistBelief [27], Petuum [28], SparkNet [29]
High-Performance Network Fabrics	S-Caffe [30], FireCaffe [31], Malt [32], COTS-HPC [33]
Comm./Comp. Overlapping	Petuum [28], Bosen [37], Poseidon [38]
ML Model Compression	Poseidon [38], CNTK [39], Bosen [37], Gupta et al. [40]
Partial Broadcasting	SFB [41], MALT [32], Mariana [42]

2.3 Network Optimization for Distributed Machine Learning

In this section, we review existing approaches for reducing the communication overhead for large-scale distributed ML.

Reduced Communication Frequency. DistBelief [27], Petuum [28] and SparkNet [29] can reduce the communication frequency for distributed ML applications. In particular, DistBelief and Petuum are designed based on the PS framework; and SparkNet uses an iterative MapReduce framework. DistBelief allows worker nodes to pull parameters every u iterations and push generated updates every v iterations, where u might not be equal to v . In Petuum, worker nodes could use cached stale parameters to compute updates, until the version of cached parameters is older than a threshold. In SparkNet, every worker node performs τ iterations of computations at each iteration, after which all parameters are aggregated, averaged and broadcasted to worker nodes.

High-Performance Network Fabrics. S-Caffe [30], FireCaffe [31], Malt [32] and COTS-HPC [33] use high-performance network fabric, such as InfiniBand, to improve the network performance of large-scale distributed ML. To improve the communication performance of GPU-GPU, several MPI implementations, such as OpenMPI, MPICH2 and MVAPICH2 provide efficient CUDA-Aware support using techniques like GPUDirect RDMA [34], [35], [36].

Communication/Computation Overlapping. Petuum [28], Bosen [37] and Poseidon [38] could overlap communication with computation. Petuum and Bosen leverage the SSP (stale synchronous parallel) synchronization model to overlap the communication of previous iterations with the computation of current iteration. Poseidon is designed for deep learning applications, which overlaps the communication of top layers with the computation of bottom layers of a convolutional neural network (CNN) in a single iteration.

ML Model Compression. Poseidon [38], CNTK [39], Bosen [37] and Gupta et al. [40] use several compression techniques to reduce the size of updates for network transmission. Specifically, Poseidon uses sufficient factors to represent dense fully connected layers of a CNN to reduce the network traffic as well as communication time. CNTK represents updates as 1-bit values for speech deep neural networks with some negative impacts on model accuracy. Bosen and Gupta et al. show that deep neural networks can be trained with 16-bit fixed-point numbers instead of 32-bit floating-point numbers.

Partial Broadcasting. SFB [41], MALT [32] and Mariana [42] uses partial broadcasting in decentralized, peer-to-peer ML systems. Native broadcasting in a peer-to-peer system would incur quadratic communication cost with respect to the number of nodes. To address this problem, SFB uses a partial broadcasting system, in which a node only broadcasts updates to a part of directly connected nodes. In MALT, nodes exchanges updates with a subset of nodes selected by a Halton sequence. Mariana manages multiple nodes workers in a linear topology, and only sends data to adjacent workers.

Discussion. Existing approaches for reducing the communication overhead of distributed ML have several limitations. First, when reducing the communication frequency, important updates cannot be pushed to the server nodes immediately, and workers nodes cannot use last parameters for computation. Second, in many

cloud computing platforms and clusters, only commodity network is available, since high-performance network fabrics could increase the infrastructure costs significantly. Third, if the communication time is much longer than computation, distributed ML applications still have quite high communication overhead even if we overlap communication with computation. Four, popular distributed ML systems usually are designed based on the PS framework, so that they cannot benefit from partial broadcasting approaches, which are designed based on a decentralized, peer-to-peer architecture.

2.4 Large-Scale Distributed Graph Processing

Pregel [43] and GAS (Gather-Apply-Scatter) [44] are two widely used vertex-centric computation models to represent distributed graph processing. In this section, we review two in-memory systems (Pregel+ [45] and PowerGraph [44]) and two out-of-core systems (GraphD [45] and Chaos [46]). We first review their graph partitioning techniques, then review their computation engines. In this section, we assume that the input graph $G = (V, E)$ contains $|V|$ vertices and $|E|$ edges. Each vertex $v \in V$ has a unique ID $id(v)$, an incoming adjacency list $\Gamma_{in}(v)$ and an outgoing adjacency list $\Gamma_{out}(v)$.

2.4.1 Graph Partitioning

Before the computation, distributed graph processing systems should divide the input graph into partitions, and assigns them to N servers. Figure 2.3 shows graph partitioning strategies used in Pregel+, PowerGraph, GraphD and Chaos.

Hash-based Edge-Cut Graph Partitioning. Pregel+ and GraphD use a hash function to assign a vertex v and its outgoing adjacency list $\Gamma_{out}(v)$ to a server, and provide interaction between vertices along out-edges by message passing. Each server approximately maintains $|V|/N$ vertices in memory. To enable in-

Figure 2.3 Graph partitioning strategies of Pregel+, PowerGraph, GraphD and Chaos. In (a) (c), each vertex is assigned to a server based on a hash function. In (b), the black circle denotes a vertex replica.

memory computation, Pregel+ needs to maintain $\Gamma_{out}(v)$ in memory, so that it requires additional memory to store $|E|$ edges during the computation. In contrast, GraphD stores $\Gamma_{out}(v)$ on disks to reduce memory footprint. This hash-based edge-cut graph partitioning strategy can evenly distribute vertices among servers, but cannot balance workloads when processing skewed graphs, since high-degree vertices need more execution time.

Intelligent Vertex-Cut Graph Partitioning. PowerGraph evenly assigns edges to servers to improve workload balance. If a server has an edge (u, v) , it should maintain vertex u and v in memory. Therefore, a single vertex may have multiple replicas on different servers. For example, in Figure 2.3 (b), both server-A and server-B maintain a replica of vertex-1. Since PowerGraph requires each vertex

v to be aware of $\Gamma_{in}(v)$ and $\Gamma_{out}(v)$, it needs double spaces to store a single edge, which is indexed by its source and target vertex separately. Therefore, during the computation, PowerGraph maintains $M|V|$ vertices and $2|E|$ edges in memory, where M is the average vertex replication factor. To reduce the storage and network overhead, many intelligent vertex-cut methods have been proposed to reduce the value of M in PowerGraph.

Streaming Partitioning. Chaos divides G into P streaming partitions, and store them on disks. Each partition consists of a set of vertices, their out-edges and received messages. All edges with the same source vertex appear in a single partition, and they are not required to be sorted. During the computation, each server processes a streaming partition at a time: it loads the vertices into memory and streams other data from disks. Therefore, each server only needs to maintain $|V|/P$ vertices in memory. It should be noted that Chaos does not use any methods to assign a streaming partition to a single server. Instead, it spreads all data of a single partition over all servers in the cluster uniformly and randomly.

2.4.2 Computation Engine

Pregel+ and GraphD follow the Pregel model. PowerGraph can process skewed graphs efficiently using the GAS model. Chaos designs an edge-centric GAS model to represent distributed out-of-core graph processing.

Pregel. A Pregel program processes the input graph in supersteps or iterations. As shown in Algorithm 2.1, in each superstep, all active vertices should execute a function *compute(msgs)* to update their values, then send messages along out-edges, and vote to halt. A halted vertex will be reactivated if it receives a message. The program terminates when there are no active vertices. To reduce the communication overhead, Pregel+ and GraphD can combine messages with the same target into a single one. Take PageRank as an example, after message

Algorithm 2.1: Pregel Abstraction

```

1. superstep = 0
2. while active_vertices  $\neq \emptyset$  do
3.   for  $v \in$  active_vertices do
4.      $v.value = v.compute(messages)$ 
5.     // send messages to corresponding vertices via network
6.      $v.send\_message(\Gamma_{out}(v))$ 
7.     // halt the vertex and activate it if it receives messages
8.      $v.halt()$ 
9.   end for
10.  superstep = superstep + 1
11. end while

```

combining, Pregel+ stores $\eta|E|$ and $|V|$ messages in memory at sender and receiver side respectively, where $0 < \eta \leq 1$ is the combining ratio. GraphD stores $|E|$ messages on disk at sender side, sends $\eta|E|$ messages over network after message combining, and digests all incoming messages in a memory buffer.

GAS. GAS represents vertex-centric computation with three phases: *gather*, *apply* and *scatter*, as shown in Algorithm 2.2. In the gather phase, each active vertex collects information along in-edges to compute an accumulator. Since a vertex can have multiple replicas, the gather function runs locally on each replica, and generates a partial result. Each mirror sends its partial result to the master, which is in charge of computing the finally accumulator. In the apply phase, the master vertex updates its value, and sends its new value to all mirrors. During the scatter phase, each vertex activates its neighbors along out-edges. Take PageRank as an example, PowerGraph keeps $M|V|$ messages in memory in the gather and apply phases, and sends $2M|V|$ messages via network in a superstep, where M is the average vertex replication factor.

Algorithm 2.2: GAS Abstraction

```

1. superstep = 0
2. while active_vertices  $\neq \emptyset$  do
3.   for  $v \in$  active_vertices do
4.     // pull data from replicas
5.     accumulator = sum( $v$ .gather( $\Gamma_{in}(v)$ ))
6.     // sync updated values to all vertex replicas
7.      $v$ .value =  $v$ .apply(accumulator,  $v$ .value)
8.     // activate neighbour vertices along out-edges
9.      $v$ .scatter( $\Gamma_{out}(v)$ )
10.  end for
11.  superstep = superstep + 1
12. end while

```

Edge-Centric GAS. Chaos designs an edge-centric GAS model to represent out-of-core graph computation with three phases: *scatter*, *gather*, and *apply*, as shown in Algorithm 2.3. During the scatter phase, for each edge, the *scatter* function computes and sends a message to its target vertex, which would be written into the associate streaming partition on disks. In the gather phase, for each message, the *gather* function update its target vertex's accumulator. Finally, during the apply phase, for each vertex, the *apply* function uses its accumulator the vertex value. Take PageRank as an example, in a single superstep, Chaos reads $2|V|$ vertices, $|E|$ edges and $|E|$ messages from disks, and writes $|E|$ messages and $|V|$ vertices into disks.

We should note that all I/O operations need network communication in Chaos, since Chaos distributes a single streaming partition across the cluster uniformly. Chaos adopts this design to balance storage across the cluster, and it assumes that cluster network bandwidth is high enough.

Algorithm 2.3: Edge-Centric GAS Abstraction

```

1. superstep = 0
2. while not done do
3.   for  $p \in \text{streaming\_partitions}$  do
4.     load_to_memory(p.vertices)
5.     for  $e \in p.\text{out\_edges}$  do scatter( $e.\text{src.value}$ ,  $e.\text{target}$ ) end for
6.   end for
7.   for  $p \in \text{streaming\_partitions}$  do
8.     load_to_memory(p.vertices)
9.     for  $m \in p.\text{messages}$  do
10.       $m.\text{target.accum} = \text{gather}(m.\text{target}, m.\text{value})$ 
11.    end for
12.    for  $v \in p.\text{vertices}$  do
13.       $v.\text{value} = \text{apply}(v.\text{accum}, v.\text{value})$ 
14.    end for
15.  end for
16.  superstep = superstep + 1
17. end while

```

2.4.3 Discussion

Existing distributed graph processing systems have several limitations when processing big graphs with billions of vertices or hundreds of billions of edges. Specifically, in-memory systems, such as Pregel+ and PowerGraph, need to store the entire graph and all intermediate data in the main memory. This strategy can easily exceed the cluster memory limit, leading to performance degradation or program crashes. While out-of-core approaches, such as GraphD and Chaos, can process big graphs from the secondary storage, the high disk I/O overhead could significantly reduce the processing performance. Thus, it motivates us to design

a new distributed graph processing system to enable high-performance big graph analytics in small clusters with limited memory.

2.5 Summary

In this section, we review existing approaches for optimizing the performance of distributed MLGP applications over virtualized infrastructure, and find that there are still a lot of challenges. First, existing CMSs could only statically allocate resources to MLGP workloads, resulting in poor cluster utilization and unfair allocation. Second, existing DHT-based metadata management systems need to process a large number of lookup operations, leading to high latency of metadata operations and reduced application performance. Third, it is important to reduce the communication overhead for large-scale distributed ML. Existing approaches require expensive infrastructures (such as InfiniBand), or only work well with a specific application rather than a general-purpose distributed ML framework. Four, existing distributed in-memory graph processing systems need to store the entire graph in main memory, resulting in memory overflow and program crashes. While out-of-core approaches can process big graphs from the secondary storage, the high disk I/O overhead could significantly reduce the processing performance. For these reasons, we design four approaches, Dorm, MetaFlow, ParameterFlow and GraphH, in this thesis to improve the performance of distributed MLGP over virtualized infrastructure.

References

- [1] Omar, Mohammed Aissaoui, and Mohsine Eleuldj. "OpenStack: toward an open-source solution for cloud computing." *International Journal of Computer Applications* 55, no. 3 (2012).
- [2] Kumar, Rakesh, Kanishk Jain, Hitesh Maharwal, Neha Jain, and Anjali Dadhich. "Apache cloudstack: Open source infrastructure as a service cloud computing platform." *Proceedings of the International Journal of advancement in Engineering technology, Management and Applied*

- Science (2014): 111-116.
- [3] Gandhi, Anshul, Sidhartha Thota, Parijat Dube, Andrzej Kochut, and Li Zhang. "Autoscaling for hadoop clusters." In Cloud Engineering (IC2E), 2016 IEEE International Conference on, pp. 109-118. IEEE, 2016.
 - [4] Vavilapalli, Vinod Kumar, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves et al. "Apache hadoop yarn: Yet another resource negotiator." In Proceedings of the 4th annual Symposium on Cloud Computing, p. 5. ACM, 2013.
 - [5] Delimitrou, Christina, and Christos Kozyrakis. "Quasar: resource-efficient and QoS-aware cluster management." In ACM SIGPLAN Notices, vol. 49, no. 4, pp. 127-144. ACM, 2014.
 - [6] Niu, Zhaojie, Shanjia Tang, and Bingsheng He. "Gemini: An adaptive performance-fairness scheduler for data-intensive cluster computing." In Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on, pp. 66-73. IEEE, 2015.
 - [7] Delimitrou, Christina, and Christos Kozyrakis. "Paragon: QoS-aware scheduling for heterogeneous datacenters." In ACM SIGPLAN Notices, vol. 48, no. 4, pp. 77-88. ACM, 2013.
 - [8] Zhang, Zhuo, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. "Fuxi: a fault-tolerant resource management and job scheduling system at internet scale." Proceedings of the VLDB Endowment 7, no. 13 (2014): 1393-1404.
 - [9] <https://docs.docker.com/engine/swarm>
 - [10] Verma, Abhishek, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. "Large-scale cluster management at Google with Borg." In EuroSys 2015, p. 18. ACM, 2015.
 - [11] Bernstein, David. "Containers and cloud: From lxc to docker to kubernetes." IEEE Cloud Computing 1, no. 3 (2014): 81-84.
 - [12] Hindman, Benjamin, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica.

- "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." In NSDI, vol. 11, no. 2011, pp. 22-22. 2011.
- [13] Ghodsi, Ali, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types." In NSDI, vol. 11, no. 2011, pp. 24-24. 2011.
- [14] Schwarzkopf, Malte, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. "Omega: flexible, scalable schedulers for large compute clusters." In EuroSys 2013, pp. 351-364. ACM, 2013.
- [15] Boutin, Eric, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. "Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing." In OSDI, vol. 14, pp. 285-300. 2014.
- [16] Delimitrou, Christina, Daniel Sanchez, and Christos Kozyrakis. "Tarcil: reconciling scheduling speed and quality in large shared clusters." In SOCC 2015, pp. 97-110. ACM, 2015.
- [17] Ousterhout, Kay, Patrick Wendell, Matei Zaharia, and Ion Stoica. "Sparrow: distributed, low latency scheduling." In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 69-84. ACM, 2013.
- [18] Delgado, Pamela, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. "Hawk: Hybrid datacenter scheduling." In Proceedings of the 2015 USENIX Annual Technical Conference, no. EPFL-CONF-208856, pp. 499-510. USENIX Association, 2015.
- [19] Karanasos, Konstantinos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. "Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters." In USENIX Annual Technical Conference, pp. 485-497. 2015.
- [20] Brandt, Scott A., Ethan L. Miller, Darrell DE Long, and Lan Xue. "Efficient metadata management in large distributed storage systems."

- In Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on, pp. 290-298. IEEE, 2003.
- [21] Stoica, Ion, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. "Chord: A scalable peer-to-peer lookup service for internet applications." *ACM SIGCOMM Computer Communication Review* 31, no. 4 (2001): 149-160.
- [22] Clements, Austin T., Dan RK Ports, and David R. Karger. "Arpeggio: Metadata searching and content sharing with Chord." In *International Workshop on Peer-To-Peer Systems*, pp. 58-68. Springer, Berlin, Heidelberg, 2005.
- [23] Gupta, Anjali, Barbara Liskov, and Rodrigo Rodrigues. "One Hop Lookups for Peer-to-Peer Overlays." In *HotOS*, pp. 7-12. 2003.
- [24] Verma, Abhishek, and Shivaram Venkataraman. *Efficient metadata management for cloud computing applications*. 2010.
- [25] DeCandia, Giuseppe, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voss, and Werner Vogels. "Dynamo: amazon's highly available key-value store." *ACM SIGOPS operating systems review* 41, no. 6 (2007): 205-220.
- [26] Lakshman, Avinash, and Prashant Malik. "Cassandra: a decentralized structured storage system." *ACM SIGOPS Operating Systems Review* 44, no. 2 (2010): 35-40.
- [27] Dean, Jeffrey, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior et al. "Large scale distributed deep networks." In *NIPS 2012*, pp. 1223-1231. 2012.
- [28] Xing, Eric P., Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. "Petuum: A new platform for distributed machine learning on big data." *IEEE Transactions on Big Data* 1, no. 2 (2015): 49-67.

- [29] Moritz, Philipp, Robert Nishihara, Ion Stoica, and Michael I. Jordan. "Sparknet: Training deep networks in spark." arXiv preprint arXiv:1511.06051(2015).
- [30] Awan, Ammar Ahmad, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K. Panda. "S-Caffe: Co-designing MPI Runtimes and Caffe for Scalable Deep Learning on Modern GPU Clusters." In Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 193-205. ACM, 2017.
- [31] Iandola, Forrest N., Matthew W. Moskewicz, Khalid Ashraf, and Kurt Keutzer. "Firecaffe: near-linear acceleration of deep neural network training on compute clusters." In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2592-2600. 2016.
- [32] Li, Hao, Asim Kadav, Erik Kruus, and Cristian Ungureanu. "Malt: distributed data-parallelism for existing ml applications." In EuroSys 2015, p. 3. ACM, 2015.
- [33] Coates, Adam, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. "Deep learning with COTS HPC systems." In International Conference on Machine Learning, pp. 1337-1345. 2013.
- [34] Potluri, Sreeram, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K. Panda. "Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs." In Parallel Processing (ICPP), 2013 42nd International Conference on, pp. 80-89. IEEE, 2013.
- [35] Shainer, Gilad, Ali Ayoub, Pak Lui, Tong Liu, Michael Kagan, Christian R. Trott, Greg Scantlen, and Paul S. Crozier. "The development of Mellanox/NVIDIA GPUDirect over InfiniBand: a new model for GPU to GPU communications." Computer Science-Research and Development 26, no. 3-4 (2011): 267-273.
- [36] Rossetti, D. A. V. I. D. E., and SW COMPUTE TEAM. "GPUDIRECT: Integrating the GPU with a Network Interface." In GPU Technology

- Conference. 2015.
- [37] Wei, Jinliang, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. "Managed communication and consistency for fast data-parallel iterative analytics." In SOCC 2015, pp. 381-394. ACM, 2015.
 - [38] Zhang, Hao, Zhiting Hu, Jinliang Wei, Pengtao Xie, Gunhee Kim, Qirong Ho, and Eric Xing. "Poseidon: A system architecture for efficient GPU-based deep learning on multiple machines." arXiv:1512.06216 (2015).
 - [39] Yu, Dong, and Li Deng. Automatic speech recognition: A deep learning approach. Springer, 2014.
 - [40] Gupta, Suyog, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. "Deep learning with limited numerical precision." In ICML-15, pp. 1737-1746. 2015.
 - [41] Xie, Pengtao, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric P. Xing. "Lighter-communication distributed machine learning via sufficient factor broadcasting." In UAI. 2016.
 - [42] Zou, Yongqiang, Xing Jin, Yi Li, Zhimao Guo, Eryu Wang, and Bin Xiao. "Mariana: Tencent deep learning platform and its applications." In VLDB Endowment 7, no. 13 (2014): 1772-1777.
 - [43] Malewicz, Grzegorz, Matthew H. Austern, Aart JC Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: a system for large-scale graph processing." In SIGMOD, pp. 135-146. ACM, 2010.
 - [44] Gonzalez, Joseph E., Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs." In OSDI, vol. 12, no. 1, p. 2. 2012.
 - [45] Yan, Da, James Cheng, Yi Lu, and Wilfred Ng. "Effective techniques for message reduction and load balancing in distributed graph computation." In Proceedings of the 24th International Conference on World Wide Web, pp. 1307-1317. International World Wide Web Conferences Steering Committee, 2015.

- [46] Yan, Da, Yuzhen Huang, James Cheng, and Huanhuan Wu. "Efficient processing of very large graphs in a small cluster." arXiv preprint arXiv:1601.05590 (2016).
- [47] Roy, Amitabha, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. "Chaos: Scale-out graph processing from secondary storage." In Proceedings of the 25th Symposium on Operating Systems Principles, pp. 410-424. ACM, 2015.

Chapter 3

Dorm: Dynamic Cluster Management for Distributed Machine Learning and Graph Processing

Many cluster management systems (CMSs) have been proposed to share a single cluster with multiple machine learning and graph processing (MLGP) workloads. However, due to their static resource allocation strategy, none of the existing approaches could achieve high cluster utilization. In this work, we propose a new CMS called Dorm, which incorporates two techniques: a dynamically-partitioned cluster management mechanism and a utilization-fairness optimizer. In particular, Dorm uses the container-based virtualization technique to partition a cluster, executes one application per partition, and dynamically resizes each partition at runtime for maintaining high cluster utilization with fairness guarantees. Extensive performance evaluations showed that Dorm could increase the cluster utilization by a factor of up to 2.32, and speed up popular distributed MLGP applications by a factor of up to 2.76, compared to existing CMSs.

3.1 Introduction

Emerging machine learning and graph processing (MLGP) applications entail the need for high performance distributed computing systems (DCSs). Specifically, many distributed ML systems have been designed based on the Parameter Server (PS) framework, such as MxNet, TensorFlow. Pregel and PowerGraph are widely used for distributed graph processing. These distributed MLGP systems usually decompose an application into a set of tasks and execute them on multiple nodes.

Aforementioned distributed MLGP systems are trending to be deployed in the same cluster, so that users can pick the most suitable system for each specific application. A standard solution is to statically partition the cluster, and run one DCS per partition. For example, we can use OpenStack or CloudStack to create a set of Virtual Machines (VMs) for MxNet and execute all MxNet applications in this virtual cluster. However, as discussed in Chapter 2, the static DCS-level resource allocation strategy can lead to low cluster utilization for two reasons:

- Popular distributed MLGP systems do not have built-in schedulers to manage and execute multiple applications at the same time. Thus, users can start one application only after the completion of the existing one.
- Busy distributed MLGP systems cannot dynamically scale out to use idle cluster resources to execute their applications.

To improve the cluster utilization, many cluster management systems (CMSs) have been proposed to statically share a cluster at the level of applications, such as Yarn and Mesos. As discussed in Chapter 2, these CMSs would allocate a static partition of the cluster to each application, and let applications reserve all acquired resources until completion. For example, when submitting a MxNet-based application to a cluster managed by Yarn, users need to specify its resource demands (e.g., the number of computation nodes, and the number of CPUs and RAM per node). Yarn would select proper resources from the cluster to fulfill

user-supplied resource demands, and uses them to run the application. However, with static application-level resource allocation strategy, distributed MLGP applications cannot dynamically scale out or in based on the global cluster state, resulting in low resource utilization.

In this work, we propose a new CMS called Dorm to handle multiple distributed MLGP workloads in a shared cluster with high cluster utilization and fairness guarantees. To achieve this goal, we introduce two techniques. First, we design a dynamically-partitioned cluster management mechanism. With this approach, Dorm gives each application a partition of the cluster, and can dynamically resize each partition without significant impacts on application performance. Second, we develop a utilization-fairness optimizer to determine optimal cluster resource allocations with cluster-wide visibility in real-time. In particular, when detecting newly submitted or completed applications, Dorm could adjust existing cluster resource allocations dynamically, and force affected applications to scale out or in accordingly. As a result, Dorm consistently maintains high resource utilization with fairness guarantees.

We implement Dorm based on Docker [1] and Swarm [2], and integrate it with six distributed MLGP systems: Petuum [3], MxNet [4], TensorFlow [5], Caffe [6], GraphH [7] and GraphD [8]. Extensive evaluations on a working testbed and large-scale simulations showed that Dorm could improve the cluster utilization by a factor of up to 2.32, and speeds up popular distributed MLGP applications by a factor of up to 2.76, compared to existing CMSs.

The remainder of this chapter is organized as follows. In section 3.2, we introduce Dorm’s system architecture and its dynamically-partitioned cluster management mechanism. Section 3.3 presents the utilization-fairness optimizer of Dorm. In Section 3.4, we introduce the implementation details of Dorm. Section 3.5 shows evaluation results. We conclude this chapter in Section 3.6.

Figure 3.1 Dorm’s system architecture. Dorm consists of a central DormMaster to manage a set of DormSlaves. Utilization-fairness optimizer is a module in DormMaster to make resource allocation decisions with cluster-wide visibility.

3.2 Dorm System Design

In this section, we introduce Dorm’s system architecture and key components, and show its dynamically-partitioned cluster management mechanism.

3.2.1 System Architecture

Figure 3.1 shows Dorm’s system architecture. Specifically, Dorm consists of a central DormMaster and a set of DormSlaves. The DormMaster would harness hardware resources (e.g., CPU, GPU, RAM) available on multiple DormSlaves to execute submitted distributed MLGP applications.

DormMaster. The DormMaster runs as a daemon on a dedicated server. It is in charge of managing all hardware resources of the cluster, and exposes them to multiple distributed MLGP applications. It uses *containers* to partition a cluster, and gives each application a partition of the cluster. The *container* is a logical

bundle of resources on a server, for example $\langle 2 \text{ CPUs}, 1 \text{ GPU}, 8\text{GB RAM} \rangle$. In Figure 3.1, the DormMaster partitions a cluster into two partitions. Each partition has four *containers*, and allocates them to two applications. In particular, the MxNet application is allocated with 8 CPUs, 4 GPUs and 16GB RAM; and the TensorFlow application has 16 CPUs and 36GB RAM. The utilization-fairness optimizer is a module in the DormMaster, which is designed to make resource allocation decisions. When detecting newly submitted applications or completed applications, it determines how to reallocate cluster resources, so that the cluster can continually keep high resource utilization with fairness guarantees. The DormMaster could enforce new resource allocations by adjusting existing ones: creating and destroying *containers* on particular servers.

DormSlave. The DormSlave runs on each cluster server, and manages the local hardware resources of the server. First, it reports the resource capacity of a server to the DormMaster. Assuming there are m types of hardware resources in server- i , let $\mathbf{c}_i = (c_{i,1}, c_{i,2}, \dots, c_{i,m})$ represent its resource capacity vector. DormSlave- i would report $\mathbf{c}_i$ to the DormMaster, which would use such information to make resource allocation decisions. For example, in Figure 3.1, DormSlave-1 reports to the DormMaster that it contains $\langle 24 \text{ CPUs}, 4 \text{ GPU}, 64\text{GB RAM} \rangle$. Second, the DormSlave uses *containers* to share a single server with multiple applications. For example, DormSlave-1 manages six *containers* belonging to two applications in Figure 3.1.

Workloads. Dorm is designed to handle multiple distributed MLGP workloads in a single cluster. As shown in Chapter 1 and Chapter 2, each distributed MLGP application relies on a particular distributed computing system like TensorFlow or MxNet, and executes its tasks on multiple worker nodes. In Dorm, each *container* provides sufficient hardware resources for a single worker node to run assigned tasks. TaskExecutor and TaskScheduler could provide required software resources on each *container* for task execution. In particular, the TaskExecutor is

the basic task execution unit, and each TaskScheduler is in charge of placing an application's tasks on the local TaskExecutor. At the beginning of an application, Dorm would automatically deploy a TaskExecutor and a TaskScheduler on each *container* of an application. For example, in Figure 3.1, Dorm automatically runs a MxNet TaskExecutor and a MxNet TaskScheduler on all *containers* allocated to App1, and runs a TensorFlow TaskExecutor and a TensorFlow TaskScheduler on all *containers* created for App2.

Container. *Containers* belonging to the same application have uniform, constant resources for two reasons. First, distributed MLGP applications can balance computation workloads across all TaskExecutors by equally partitioning the input datasets. Second, distributed MLGP applications usually use iterative algorithms to process the input datasets without changing resource demands during application runtime. For example, in Figure 6, all *containers* of App1 contains 2 CPUs, 1GPU and 4GB RAM.

Application Submission. To submit a new application to Dorm, users need to specify its resource demands by providing a 6-tuple as follows:

$$(executor, \mathbf{d}, w, n_{max}, n_{min}, cmd),$$

where *executor* is a string (e.g., “MxNet” or “TensorFlow”) to indicate the name of the required distributed MLGP system; $\mathbf{d}$ is the resource demand vector (e.g., $\langle 2 \text{ CPUs}, 1 \text{ GPU}, 8 \text{GB RAM} \rangle$) per *container*; w is an integer to denote this application's weight; n_{max} and n_{min} represent the maximum and minimum numbers of *containers* this application could have; *cmd* specifies the scripts used to start and resume this application. The utilization-fairness optimizer determines when to launch these newly submitted applications, and allocate suitable cluster resources to them based on the algorithms detained in Section 3.3.

Container-based Resource Isolation. Dorm uses container-based virtualization techniques to provide performance isolation between different applications in the

same cluster. Several container techniques have been proposed, such as Linux Containers (LXC) [9], OpenVZ [10] and Docker [1]. Compared to Virtual Machines, containers could limit CPU, RAM, network, I/O usages of a process tree with much lower overhead. Existing CMSs, such as Mesos and Yarn, also use containers for resource isolation. In Dorm, we use Docker to partition the cluster, and each Dorm *container* contains a set of resources isolated by Docker.

3.2.2 Dynamically-Partitioned Resource Allocation

Dorm allocates cluster resources to distributed MLGP workloads dynamically at the level of applications. In a nutshell, it gives each application a partition of the cluster, and can dynamically resize each partition during the application runtime.

Making Allocation Decisions. When new applications are submitted, or existing applications finish, the utilization-fairness optimizer decides how to repartition the cluster with cluster-wide visibility. In this way, Dorm can consistently achieve high resource utilization and low fairness loss. Since Dorm uses the *container* as the basic resource allocation unit, the utilization-fairness optimizer determines the number of *containers* offered to each application and the location of each container based on the algorithm detailed in Section 3.3.

Adjusting Existing Allocations. Dorm could enforce new resource allocations by creating and destroying *containers* on particular servers. However, modern distributed MLGP applications cannot automatically take advantage of newly acquired resources, or continue to run with revoked resources. To address this problem, we propose a checkpoint-based resource adjustment protocol. More specifically, when adjusting an application’s resources, Dorm would firstly save its state to a reliable storage system (e.g., the Lustre file system). Then, Dorm kills this application and creates/destroys *containers* on corresponding servers. Finally, Dorm resumes the killed application from the saved state with new

Figure 3.2 An example to show how Dorm performs resource allocation. In this example, there are two running applications in the cluster. When a user submits the third application, Dorm adjusts existing resource allocations to run the new application.

resources. In this way, distributed MLGP applications can dynamically scale in or out without re-computing from the first iteration.

Resource adjustment would not have a significant impact on distributed MLGP application performance. Based on our measurements using various distributed MLGP systems and applications, the average time used to kill and resume an application is about 37s, which is affected by application state size, network bandwidth, disk I/O, etc. Compared to the long application duration (e.g., 4 to 20 hours), the resource adjustment overhead would not significantly reduce the application performance. Moreover, when making resource allocation decisions, the utilization-fairness optimizer tries to minimize the number of affected applications, which is detailed in Section 3.3.

An Example. Figure 3.2 shows an example of how Dorm works. In step (1), a user submits a new application to Dorm with following information:

$$\begin{aligned} \text{executor} &= \text{"Caffe"}, \\ \mathbf{d} &= \langle 1 \text{ CPU}, 1 \text{ GPU}, 8 \text{ GB RAM} \rangle, \end{aligned}$$

Table 3.1 Summary of notations used in Dorm.

$x_{i,j}^t$	application i 's container number on DormSlave j at time t
l_i^t	application i 's fairness loss at time t
r_i^t	application i 's resources are adjusted at time t
u_i^t	resource i 's utilization at time t
b	number of DormSlaves
m	number of resource types
w_i	application i 's weight
n_i^{max}	application i 's maximum container number
n_i^{min}	application i 's minimum container number
θ_1	threshold of fairness loss
θ_2	threshold of resource adjustment overhead
$\hat{s}_i^t$	application i 's theoretical resource share based on DRF at time t
s_i^t	application i 's actual resource share at time t
$d_{i,j}$	application i 's resource demand per container on resource j
$c_{i,j}$	DormSlaves i 's resource capacity on resource j
B	set of DormSlaves, $B = \{1, 2, \dots, b\}$
M	set of resource types, $M = \{1, 2, \dots, m\}$
A_r^t	set of running applications at time t
A_p^t	set of pending applications at time t
A_n^t	set of newly launched applications at time t
A_c^t	set of completed applications at time t

$$\begin{aligned}
w &= 2, \\
n_{max} &= 5, \\
n_{min} &= 1, \\
cmd &= ["start.sh", "resume.sh"].
\end{aligned}$$

In step (2), the utilization-fairness optimizer determines that all applications have two containers on DormSlave-1. In step (3), the DormMaster enforces new resource allocations by destroying two *containers* of App2 and creating two *containers* for the new application. During this step, Dorm saves App2's state to a reliable storage system and kill it. In step (4), the DormMaster configures TaskExecutors and TaskSchedulers on new *containers*, starts App3, and resumes App2. In step (5), the DormMaster returns App3's status to the user.

3.3 Utilization-Fairness Optimizer

When new applications are submitted, or existing applications finish at time t , Dorm would re-allocate cluster resources to the running application set A_r^t . In this section, we show how the utilization-fairness optimizer makes resource allocation decisions to improve the cluster utilization with fairness guarantees. Table 3.1 shows used symbols and their definitions in this section.

3.3.1 Objectives

We consider three objectives when allocating resources to the running application set A_r^t at time t : high cluster resource utilization, low fairness loss and low resource adjustment overhead. A_r^t can be represented as follows:

$$A_r^t = A_r^{t-1} \setminus A_c^t \cup A_n^t, \quad (3.1)$$

where A_r^{t-1} is the set of application running at time $t - 1$; A_c^t is the set of completed applications at time t ; and A_n^t is the set of newly launched applications at time t .

Objective 1: High Cluster Utilization. The cluster's utilization is defined as the sum of all m types of hardware resources' utilization at time t , which can be represented as follows:

$$\begin{aligned} ClusterUtilization &= \sum_{k \in M} u_k^t \\ &= \sum_{k \in M} \sum_{i \in A_r^t} \sum_{j \in B} \frac{x_{i,j}^t d_{i,k}}{\sum_{h \in B} c_{h,k}} \end{aligned} \quad (3.2)$$

where $\sum_{h \in B} c_{h,k}$ denotes the total amount of resource k in the cluster, and $x_{i,j}^t d_{i,k}$ is the amount of resource k allocated to application i on slave node j at time t .

Objective 2: Low Fairness Loss. Fairness indicates that each application could receive a fair share of resources. In this work, we use dominant resource fairness

(DRF) [11] as the fairness policy. DRF seeks to maximize the minimum dominant share across all applications. In particular, the dominant resource is the most heavily demanded resource of an application, and dominant share is the share of the dominant resource. Let $\hat{s}_i^t$ denote the dominant share of application i derived from DRF based on the algorithms shown in [11] based on its resource requirements and weight. Let s_i^t denote application i 's actual dominant share. Application i 's fairness loss at time t is $l_i^t = |s_i^t - \hat{s}_i^t|$. The cluster's fairness loss is the sum of all applications' fairness loss, which can be represented as follows:

$$\begin{aligned}
 \text{FairnessLoss} &= \sum_{i \in A_r^t} l_i^t \\
 &= \sum_{i \in A_r^t} |s_i^t - \hat{s}_i^t| \\
 &= \sum_{i \in A_r^t} \left| \max_{k \in M} \left(\frac{d_{i,k} \sum_{j \in B} x_{i,j}^t}{\sum_{h \in B} c_{h,k}} \right) - \hat{s}_i^t \right|,
 \end{aligned} \tag{3.3}$$

where $\max_{k \in M} (d_{i,k} \sum_{j \in B} x_{i,j}^t / \sum_{h \in B} c_{h,k})$ is application i 's actual dominant share.

Objective 3: Low Resource Adjustment Overhead. The cluster's resource adjustment overhead is measured by the number of affected applications, which would be killed and resumed, when adjusting existing resource allocations. Let r_i^t denote whether Dorm would affect application i :

$$r_i^t = \begin{cases} 0, & \text{if } x_{i,j}^{t-1} = x_{i,j}^t, \forall j \in B \\ 1, & \text{if } x_{i,j}^{t-1} \neq x_{i,j}^t, \exists j \in B \end{cases}, \forall i \in A_r^t \cap A_r^{t-1}, \tag{3.4}$$

where $A_r^t \cap A_r^{t-1}$ denotes the set of applications running at both time $t - 1$ and t . If $r_i^t = 0$, Dorm would not create or destroy *containers* on any servers for application i , and vice versa. The cluster's resource adjustment overhead can be represented as follows:

$$\text{AdjustmentOverhead} = \sum_{i \in A_r^t \cap A_r^{t-1}} r_i^t. \tag{3.5}$$

3.3.2 Making Resource Allocation Decisions

Assuming we have known the newly launched application set A_n^t (detailed in in Section 3.2.3) at time t , we describe how Dorm determines the number of *containers* offered to applications, and location of each *container*. We formulate this problem as a multi-objective optimization problem:

$$\mathbf{P3.1:} \quad \max \left[\underbrace{\sum_{i \in M} u_i}_{\text{Cluster Utilization}}, \underbrace{-\sum_{i \in A_r^t} l_i}_{\text{Fairness Loss}}, \underbrace{-\sum_{i \in A_r^t \cap A_r^{t-1}} r_i^t}_{\text{Adjustment Overhead}} \right] \quad (3.6)$$

$$\text{s.t.} \quad \sum_{i \in A_r^t} x_{i,j}^t d_{i,k} \leq c_{j,k}, \quad \forall k \in M, \forall j \in B \quad (3.7)$$

$$\sum_{j \in B} x_{i,j}^t \leq n_i^{\max}, \quad \forall i \in A_r^t \quad (3.8)$$

$$\sum_{j \in B} x_{i,j}^t \geq n_i^{\min}, \quad \forall i \in A_r^t \quad (3.9)$$

$$x_{i,j}^t \in \mathbb{Z}_0^+, \quad \forall i \in A_r^t, \forall j \in B \quad (3.10)$$

Equation 3.6 is the objective function, which shows that we want to maximize resource utilization, minimize fairness loss and minimize resource adjustment overhead. We have several constraints. Specifically, Equation 3.7 indicates that each cluster server cannot exceed its resource capacity. Equation 3.8 and 3.9 constraint the maximum and minimum numbers of containers an application can have. Equation 3.10 shows that $x_{i,j}^t$ is an integer variable.

We then use the ε -constraints method [12] to transform the problem **P3.1** into a typical mixed-integer linear programming (MILP) problem as follows:

$$\mathbf{P3.2:} \quad \max \sum_{k \in M} \sum_{i \in A_r^t} \sum_{j \in B} \frac{x_{i,j}^t d_{i,k}}{\sum_{h \in B} c_{h,k}} \quad (3.11)$$

$$\text{s.t.} \quad l_i^t \geq s_i^t - \hat{s}_i^t, \quad \forall i \in A_r^t \quad (3.12)$$

$$l_i^t \geq \hat{s}_i^t - s_i^t, \quad \forall i \in A_r^t \quad (3.13)$$

$$Mr_i^t \geq x_{i,j}^{t-1} - x_{i,j}^t, \quad \forall i \in A_r^t \cap A_r^{t-1}, \forall j \in B \quad (3.14)$$

$$Mr_i^t \geq x_{i,j}^t - x_{i,j}^{t-1}, \quad \forall i \in A_r^t \cap A_r^{t-1}, \forall j \in B \quad (3.15)$$

$$l_i^t \in \mathbb{R}_0^+, \quad \forall i \in A_r^t \quad (3.16)$$

$$r_i^t \in \{0, 1\}, \quad \forall i \in A_r^t \quad (3.17)$$

$$\sum_{i \in A_r^t} l_i^t \leq [\theta_1 \times 2m], \quad (3.18)$$

$$\sum_{i \in A_r^t} r_i^t \leq [\theta_2 \times |A_r^t|], \quad (3.19)$$

$$(3.7) (3.8) (3.9) (3.10)$$

In **P3.2**, we choose resource utilization as the objective to be maximized; fairness loss and resource adjustment overhead are constrained to be no greater than some given thresholds. Equation 3.12 and 3.13 are used to linearize l_i^t . Equation 3.14 and 3.15 are used to linearize r_i^t with a big number M . Equation 3.18 and 3.19 are the constraints for fairness loss and adjustment overhead with threshold θ_1 and θ_2 , where $\theta_1 \in [0, 1]$ and $\theta_2 \in [0, 1]$. Specially, if $\theta_1 = 1$, we would try to maximize the cluster utilization without considering fairness. $\theta_1 = 0$ means that all resources are allocated strictly based on DRF. $\theta_2 = 1$ means that we would not consider any resource adjust overhead. In this case, we may kill and resume all applications during the resource adjustment operation. When $\theta_2 = 0$, it is not allowed to kill and resume any running applications when during the resource adjustment operation.

Equation 3.12 and 3.13 could linearize l_i^t in the following ways. In **P3.1**, one of our objectives is to minimize fairness loss: $\min \sum_{v \in A_r^t} l_i^t$, and $l_i^t = |s_i^t - \hat{s}_i^t|$. To implement this problem as a linear programming problem in CPLEX, we define $l_i^t \geq s_i^t - \hat{s}_i^t$ and $l_i^t \geq \hat{s}_i^t - s_i^t$ as two constraints. We then get following problem: $\min \sum_{v \in A_r^t} l_i^t$, constrained by $l_i^t \geq s_i^t - \hat{s}_i^t$ and $l_i^t \geq \hat{s}_i^t - s_i^t$, $\forall i \in A_r^t$. In this problem, if all constraints are satisfied, $l_i^t \geq |s_i^t - \hat{s}_i^t|$. Since our objective is to minimize $\sum_{v \in A_r^t} l_i^t$, the problem solver would return $l_i^t = |s_i^t - \hat{s}_i^t|$. In this way,

Equation 3.12 and 3.13 linearize l_i^t without affecting optimal solution. Equation 3.14 and 3.15 could linearize r_i^t using the same way.

P3.2 is a standard MILP problem with $(2 + b)|A_r^t|$ variables, where $x_{i,j}^t$ is an integer variable, l_i^t is a continuous variable and r_i^t is a binary variable. The objective function and constraints are all linear functions. **P3.2** is NP-hard. To see this, we assume that there are no constraints on fairness loss or adjustment overhead, we further assume that the minimum container number of application is zero. In this case, **P3.2** is can be reduced to a bounded knapsack problem. Since the bounded knapsack problem is NP-hard [14], unless $P = NP$, it is not possible to develop an algorithm which can solve **P3.2** in polynomial time.

As it is a standard MILP problem, it can be efficiently solved by algorithms like branch and cut [15], and standard MILP solves such as CPLEX. By solving **P2**, we can obtain optimal resource allocations. Then, Dorm could adjust existing resource allocations to consistently keep high utilization with fairness guarantees. If $x_{i,j}^t - x_{i,j}^{t-1} > 0$, Dorm should create $x_{i,j}^t - x_{i,j}^{t-1}$ containers for application i on DormSlave j . If $x_{i,j}^t - x_{i,j}^{t-1} < 0$, Dorm destroys $x_{i,j}^{t-1} - x_{i,j}^t$ containers for application i on DormSlave j .

3.3.3 Determining Newly Launched Applications

All submitted applications are first stored in the pending application queue A_p^t . Recalled that $A_r^t = A_r^{t-1} \setminus A_c^t \cup A_n^t$. Dorm should determine when to launch an application from the pending queue, and the set of newly launched applications set A_n^t at time t . To avoid starving pending applications, Dorm tries to maximize the number of newly launched applications at a time. Algorithm 3.1 shows how Dorm determines the newly launched application set A_n^t . specifically, Dorm iteratively selects the oldest pending application from A_p^t and appends it to A_n^t

Algorithm 3.1: Determining Newly Launched Applications**Input:** A_p^t **Output:** A_n^t

1. **while** A_p^t is not empty **do**
2. *// Find the oldest pending application*
3. $APP = \text{head}(A_p^t)$
4. $A_n^t \leftarrow A_n^t \cup \{APP\}$
5. **if** **P2** has feasible solutions **do**
6. $A_p^t \leftarrow A_p^t \setminus \{APP\}$ *// Launch App at time t*
7. **else**
8. $A_n^t \leftarrow A_n^t \setminus \{APP\}$, and break the loop
9. **end if**
10. **end while**
11. **return** A_n^t

until A_p^t is empty. At each iteration, if the problem **P2** has any feasible solutions, Dorm marks the selected pending application as a newly launched application and deletes it from A_p^t . Otherwise, Dorm deletes it from A_n^t and break the loop. Algorithm 3.1 returns A_n^t . Then, we would use A_n^t to construct A_r^t , use it as the input of **P2**, and find the optimal solution of it. Finally, Dorm would enforce the new resource allocations by creating and destroying *containers* accordingly.

3.4 Implementation Details

We implement Dorm using two open-source projects: Docker and Swarm. In particular, Docker leverages container-based virtualization techniques to provide resource isolation. Swarm provides the ability to transparently use the standard Docker APIs to create and destroy *containers* on multiple servers.

Figure 3.3 Dorm implementation details.

Resource Isolation. We consider three types of resources: CPU, GPU, and RAM.

- We use the *cpuset* flag of Docker APIs to allocate CPUs to *containers*. For example, *-cpuset-cpus=1,3* indicates that the *container* can use CPU 1, 3 on the server.
- We use the *device* flag of Docker APIs to specify the GPUs allocated to a *container*. For example, *-device=/dev/nvidia1:/dev/nvidia1* indicates that the *container* could use NVIDIA GPU 1 for computation.
- We use the *memory* flag of Docker APIs to limit RAM size of a *container*. For example, *-memory=2G* means that a *container* can use up to 2GB RAM.

Application Automation. Dorm launches applications in three steps. In the first step, the utilization-fairness optimization (which is implemented using Python and CPLEX) makes resource allocation decisions. In the second step, Dorm creates *containers* on specified cluster servers, and deploys a TaskExecutor and a TaskScheduler on each *container*. In our implementation, we set up a Docker registry to centrally manage all software resources (e.g., MxNet, TensorFlow) in the form of Docker images. When creating new *containers*, Dorm needs to clone specific Docker images. In this way, all *containers* would have all needed tools and libraries for task execution. In the third step, Dorm automatically configures

the operating systems and software of all *containers*, and starts the application. For example, Dorm could configure the IP address of each *container*, and lets *containers* belonging to the same application be aware of each other's IP address.

RESTful APIs. We use Flask to implement RESTful APIs for users to submit applications, query job status, and monitor cluster status. In particular, users could use the POST method to submit a new job with 6-tuple as follows:

```
POST /Dorm/NewJob HTTP/1.1
Content-Type: application/json
{
    "executor": "MxNet",
    "resource_demand": [2, 1, 6],
    "weight": 1,
    "n_max": 10,
    "n_min": 1,
    "script": ["sh start.sh", "sh resume.sh"]
}
```

Fault Tolerant. Dorm stores all crucial data in etcd, which is a distributed key-value storage system. Specifically, each server reports its resource capacity and resource usage (e.g., the number of *containers*) to etcd. The DormMaster uploads all application information (e.g., the submitted 6-tuple) to etcd. In this way, even the DormMaster crashes, a new DormMaster could transparently manage all resources and applications in this cluster.

3.5 Performance Evaluations

In this section, we investigate Dorm's performance using a testbed and large-scale simulations with generate online and offline workloads.

3.5.1 Experiments' Parameters and Configurations

Testbed Setup. The testbed contains 21 computation servers and 2 storage servers, which are connected by 10Gbps Ethernet. In the experiments, we use 20 computation servers as DormSlaves, each of which contains 2 Intel E5-2620 processors, 128GB RAM and 4TB HDD. Five computation servers have a NVIDIA K40 GPU per server. The DormMaster is deployed on a dedicated computation server, which contains 2 Intel E5-2620 processors, 64GB RAM and 256GB SSD. All training datasets are stored on the two storage servers. In this testbed, the DormMaster manages 240 CPU cores, 5 GPUs and 2.5TB RAM.

Simulation Setup. To evaluate Dorm in a large cluster, we use Python to design a simulator, and create a virtual cluster on it based on the scale of a production cluster in Sensetime Group Limited. Specifically, the simulator contains 200 servers, each of which contained 12 CPUs, 4 GPUs, and 128GB RAM. In the simulator, if a CPU core is occupied, then its utilization is 100%. Similarly, if a GPU is used for computation, its utilization is treated as 100%. Also, we assume that there is no network congestion problem in the simulator. For example, if an application uses 1 hour to finish in our testbed, we assume that this application could also finish after 1 hour in the simulator using the same amount of resources. Additionally, applications have linear speedup ratio in the simulator. For example, if an application could finish in 2 hours with two containers, it could finish in 1 hour if it has four containers. Since the simulator cannot factually execute applications, we should specify the estimated execution time of each application when using only one container to run it.

Dorm Configuration. We use following configurations of Dorm. Recall that θ_1 is the threshold of fairness loss, and θ_2 is the threshold of adjustment overhead.

Dorm-1	Configuration 1	$\theta_1 = 0.2$	$\theta_2 = 0.1$
Dorm-2	Configuration 2	$\theta_1 = 0.1$	$\theta_2 = 0.2$
Dorm-3	Configuration 3	$\theta_1 = 0.1$	$\theta_2 = 0.1$

Table 3.2 Applications used for synthetic workloads in the testbed.

Type	System	Datasets	Application	Resource Demands	Weight	$n_{\max}$	$n_{\min}$	Num
1	MxNet	Criteo	LR	$\langle 2 \text{ CPUs}, 0 \text{ GPU}, 8 \text{ GB RAM} \rangle$	1	32	1	20
2	GraphH	UK-2014	PageRank	$\langle 2 \text{ CPUs}, 0 \text{ GPU}, 6 \text{ GB RAM} \rangle$	2	32	1	20
3	Caffe	CIFAR-10	CaffeNet	$\langle 4 \text{ CPUs}, 0 \text{ GPU}, 6 \text{ GB RAM} \rangle$	4	8	1	6
4	MxNet	ImageNet	VGG	$\langle 4 \text{ CPUs}, 1 \text{ GPU}, 32 \text{ GB RAM} \rangle$	1	5	1	1
5	TensorFlow	ImageNet	GoogLeNet	$\langle 6 \text{ CPUs}, 1 \text{ GPU}, 16 \text{ GB RAM} \rangle$	1	5	1	1
6	Petuum	ImageNet	AlexNet	$\langle 6 \text{ CPUs}, 1 \text{ GPU}, 16 \text{ GB RAM} \rangle$	2	5	1	1
7	Caffe	ImageNet	ResNet-50	$\langle 4 \text{ CPUs}, 1 \text{ GPU}, 32 \text{ GB RAM} \rangle$	4	5	1	1

Figure 3.4 Distribution of application duration using single container in simulation.

Workloads. We generate an online workload and an offline workload for the testbed and the simulation based on the workload model of a production cluster in Sensetime Group Limited.

In the testbed, we use seven types of distributed MLGP applications to generate an online workload and an offline workload, as shown in Table 3.2. Specifically, 20 MxNet applications train logistic regression (LR) models using Criteo’s click logs as the training dataset. 20 GraphH applications perform PageRank on the UK-2014 dataset. 6 Caffe applications train CaffeNet on CIFAR-10 datasets. The

last four applications use ImageNet as the training datasets, and train VGG-16, GoogLeNet, AlexNet, and ResNet-50 with GPU, respectively. To generate the online workload, we randomly submit the 50 applications to Dorm with a mean interval time of 20 minutes. To generate the offline workload, we submit all 50 applications at the beginning of the experiments.

In the simulations, we synthesize a workload with 500 applications based on the distributed MLGP application duration distribution collected from the testbed. The application duration with single container is shown in Figure 3.4. In the simulator, we generate the estimated application execution time with single container based on the shown distribution. In particular, 90% of applications run more than 6 hours. To generate the online workload, we randomly submit the 500 applications with a mean interval time of 20 minutes. To generate the offline workload, we submit all 500 applications at the beginning of the experiments.

Baseline System. We use Swarm as the baseline system, which allocates cluster resources statically at the level of applications. Specifically, it creates 8, 8, 4, 2, 2, 2, 3 containers to run the 7 types of applications in Table 3.2, and place them in the cluster using the multidimensional bin packing algorithm [13].

3.5.2 Cluster Performance with Online Workloads

Resource Utilization. Compared to the baseline system, Dorm considerably improves the cluster resource utilization in testbed and simulations with online workloads. The performance gain comes from the dynamic resource allocation ability of Dorm, since MLGP applications could automatically scale out or in and leverage idle resources for computation. Figure 3.5 (a) (b) (c) (d) show the cluster resource utilization in the testbed with online workloads. Figure 3.5 (e) (f) (g) (h) show the results obtained using simulations.

Figure 3.5 Cluster resource utilization with online workloads in testbed (a, b, c, d) and simulation (e, f, g, h). The red line is the baseline system's resource utilization.

In the first 5 hours, the baseline system has quite low resource utilization (which is up to 1.8) on the testbed, since it could only statically allocate resources to the first 15 submitted applications. When using Dorm, these submitted applications could dynamically scale out to take advantage of idle resources. In particular, Dorm-1, Dorm-2 and Dorm-3 can increase the resource utilization by a factor of 2.55, 2.46 and 2.32 on average in the first 5 hours, respectively. From the 8th hour to the 16th hour, the baseline system runs about 20 applications, and has higher resource utilization (which is 2.38 on average) than the first 5 hours. In this case, Dorm-1, Dorm-2 and Dorm-3 could increase the resource utilization by a factor of 1.17, 1.10 and 1.08 on average, since it can reduce resource segmentation by dynamically adjusting the amount and the location of resources allocated to each application. From the 16th hour to the 24th hour, we can observe that the resource utilization of the baseline system becomes low again (which is 1.26 on average) and takes about 23 hours to complete all 50 applications, since the uncompleted applications cannot use idle resources for computation. As a comparison, Dorm-1, Dorm-2 and Dorm-3 take less than 20 hours to complete the 50 applications.

Results obtained using simulations, as shown in Figure 3.5 (e) (f) (g) (h), confirm Dorm's resource utilization improvement in large clusters with online workloads. In particular, in the first 5 hours, Dorm-1, Dorm-2 and Dorm-3 can increase the resource utilization by a factor of 2.76, 2.55 and 2.38 on average, respectively. From the 8th hour to the 12th hour, the corresponding cluster resource utilization improvement factors are 1.22, 1.19 and 1.07. Due to Dorm's dynamic resource allocation ability, Dorm-1, Dorm-2 and Dorm-3 could complete all applications within 17 hours, while the baseline system roughly takes 24 hours.

Fairness Loss. Dorm limits the fairness loss within a threshold, and can tolerate higher fairness loss with a larger θ_1 with online workloads. From Figure 3.6 (a) (b), we can see that Dorm-1 and Dorm-3, which set θ_1 to 0.2 and 0.1 with the same θ_2 , can limit the fairness loss within 1.2 and 0.6 in the testbed, respectively.

Figure 3.6 Fairness loss with online workloads. In this set of experiments, we adjust the value of θ_1 (the threshold of fairness loss) and use the same value of θ_2 (the threshold of resource adjustment overhead).

The corresponding value of the baseline system is 1.0. While Dorm-1 provides higher resource utilization than the baseline system and Dorm-3, its fairness loss is about 1.2 times higher than the baseline system from the 8th hour to the 12th hour. As a comparison, Dorm-3 could reduce the fairness loss by a factor of 1.8 on average in the testbed, compared to the baseline system.

Simulation results are similar to testbed with regard to the fairness loss. As shown in Figure 3.6 (c) (d), the baseline system's fairness loss is less than 0.7 with online workloads. The corresponding values of Dorm-1 and Dorm-3 are 1.1 and 0.3. We can observe that Dorm could provide higher cluster resource utilization at the cost of higher fairness loss by using a large θ_1 (e.g., Dorm-1). When using a small θ_1 (e.g., Dorm-3), Dorm can guarantee that resources are fairly allocated to actively applications based on DRF. It should be noted that while Dorm-3 offers

Figure 3.7 Resource adjustment overhead with online workloads. In this set of experiments, we adjust the value of θ_2 (the threshold of resource adjustment overhead) and use the same value of θ_1 (the threshold of fairness loss).

lower fairness loss than the baseline system, it still provides higher resource utilization than the baseline system.

Resource Adjustment Overhead. From Figure 3.7, we can see that Dorm can limit the resource adjustment overhead within a threshold, and can tolerate higher resource adjustment overhead with a larger θ_2 . In the testbed, Dorm-2, which set θ_2 to 0.2, reallocates cluster resources 87 times. Specifically, Dorm-2 kills and resumes 2 applications at most per operation, and affects 80 applications. When setting θ_2 to 0.1 without changing θ_1 , Dorm-3 kills and resumes 76 applications in the testbed. In the simulations, Dorm-2 kills and resumes 5 applications at most, and affects 165 applications in total. When using a smaller θ_2 , Dorm-3 affects 124 applications in total.

Figure 3.8 Cluster resource utilization with offline workloads in testbed (a, b, c, d) and simulation (e, f, g, h). The red line is the baseline system's resource utilization.

3.5.3 Cluster Performance with Offline Workloads

Resource Utilization. Due to our proposed dynamic cluster resource allocation mechanism for distributed MLGP, Dorm could considerably improve the cluster resource utilization with offline workloads in our testbed and simulations.

Figure 3.8 (a) (b) (c) (d) show the cluster resource utilization in the testbed with offline workloads. On average, in the first 6 hours, Dorm-1, Dorm-2 and Dorm-3 can improve the cluster resource utilization by a factor to 1.19, 1.11 and 1.09, respectively. In particular, the resource utilization of the baseline system is about 2.35 on average in the first 12 hours. The corresponding values of Dorm-1, Dorm-2 and Dorm-3 are 2.79, 2.62, and 2.55, respectively. From the 8th hour to the 11th hour, the baseline system's cluster resource utilization is about 0.95, since it can only statically allocate resources to a small set of applications. As a comparison, at the 8th hour, Dorm-1, Dorm-2 and Dorm-3 can auto-scale the uncompleted applications, improving the cluster utilization to 1.25. We can see that Dorm-1, Dorm-2 and Dorm-3 could reduce the total time used to complete all applications by a factor of around 1.25 in the testbed.

As shown in Figure 3.8 (e) (f) (g) (h), Dorm can also improve cluster resource utilization in the simulations. In the first 4 hours, Dorm-1, Dorm-2 and Dorm-3 could improve the cluster resource utilization by a factor of 1.12, 1.07 and 1.08 respectively, since Dorm can reduce the resource segmentation. After the 6th hour, the baseline system's cluster resource utilization is less than 1, since it cannot let uncompleted applications take advantage of idle resources for computation. As a comparison, Dorm can automatically scale-out uncompleted applications, and continuously keep high resource utilization. In particular, Dorm-1, Dorm-2 and Dorm-3 could roughly reduce the total time used to complete all applications by a factor of 1.21 in the simulations with offline workloads.

Figure 3.9 Fairness loss with offline workloads. In this set of experiments, we adjust the value of θ_1 (the threshold of fairness loss) and use the same value of θ_2 (the threshold of resource adjustment overhead).

Fairness Loss. When processing offline workloads, Dorm can limit the fairness loss within a threshold, and can tolerate higher fairness loss with a larger θ_1 , as shown in Figure 3.9. In the testbed, Dorm-1 and Dorm-3, which set θ_1 to 0.2 and 0.1, can limit the fairness loss within 1.2 and 0.6 respectively. The corresponding value of the baseline system is 1.0. In the simulations, the baseline system's fairness loss is less than 0.7 with online workloads. The corresponding values of Dorm-1 and Dorm-3 are 1.2 and 0.3. We can observe that Dorm-1 has higher fairness loss than the baseline system, since it tries to improve the cluster resource utilization with a large θ_1 . Compared to the baseline system, Dorm-3 reduces the fairness loss by a factor of 1.52 on average in both the testbed and the simulations with a small θ_1 .

Figure 3.10 Resource adjustment overhead with offline workloads. In this set of experiments, we adjust the value of θ_2 (the threshold of resource adjustment overhead) and use the same value of θ_1 (the threshold of fairness loss).

Resource Adjustment Overhead. From Figure 3.10, we can observe that Dorm can limit the resource adjustment overhead within a threshold, and can tolerate higher resource adjustment overhead with a larger θ_2 when processing offline workloads. In the testbed, Dorm-2, which set θ_2 to 0.2, reallocates cluster resources 56 times, kills and resumes 6 applications at most during a single operation, and affects 134 applications in total. When setting θ_2 to 0.1, Dorm-3 affects 103 applications in total. Simulation results are similar with testbed results. As shown in Figure 2.10 (c), Dorm affects at most 18 applications during a single resource allocation operation, and affects 1131 applications in total (Dorm may adjust some applications' resource allocations several times). The corresponding values of Dorm-3 are 9 and 283.

Figure 3.11 Application speedup ratio with online and offline workloads.

3.5.4 Application Speedup

Figure 3.11 (a) shows that Dorm-1, Dorm-2 and Dorm-3 can speed up distributed MLGP applications by a factor of 3.01, 2.78 and 2.76 on average in the testbed with online workloads, respectively. The corresponding speedup ratios in the simulation are 3.21, 2.96, and 3.05. From Figure 3.11 (b), we can observe that Dorm could improve the performance of distributed MLGP applications with offline workloads. More specifically, Dorm-1, Dorm-2 and Dorm-3 can speed up distributed MLGP applications by a factor of 1.62, 1.51 and 1.49 on average in the testbed. The corresponding values in the simulation are 2.03, 1.83 and 1.77.

The performance gain comes from improved cluster resource utilization. More specifically, when using Dorm, distributed MLGP applications can dynamically scale out to leverage idle cluster resources to execute their tasks. While Dorm needs to kill and resume running applications when adjusting existing resource allocations, such resource adjustment overhead is acceptable when considering the performance improvement caused by the improved cluster utilization.

3.6 Summary

In this work, we propose a novel cluster management system named Dorm to share a cluster with distributed MLGP workloads efficiently and fairly. Existing

cluster management systems can only allocate a static partition of the cluster to each distributed MLGP application, leading to poor resource utilization. We tackle this problem by introducing two techniques in Dorm: a dynamically-partitioned cluster management mechanism and a utilization-fairness optimizer. Each distributed MLGP application runs in a dynamic partition of the cluster. When detecting newly submitted or completed applications, Dorm's utilization-fairness optimizer makes new resource allocation decisions, and reallocates existing cluster resources accordingly to consistently keep high utilization and low fairness loss. We have implemented Dorm and enabled it to support Petuum, MxNet, TensorFlow, Caffe and GraphH. Extensive performance evaluations have shown that Dorm could simultaneously increase the resource utilization by a factor of up to 2.32, reduce the fairness loss by a factor of up to 1.52, and speed up popular distributed MLGP applications by a factor of up to 2.76, compared to existing approaches. In the future, we will integrate Dorm with more distributed MLGP systems, and make it support multiple classes of MLGP applications with different properties.

References

- [1] Merkel, Dirk. "Docker: lightweight linux containers for consistent development and deployment." *Linux Journal* 2014, no. 239 (2014): 2.
- [2] <https://docs.docker.com/engine/swarm>
- [3] Xing, Eric P., Qirong Ho, Wei Dai, Jin-Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. "Petuum: A New Platform for Distributed Machine Learning on Big Data." In *Proceedings of the 21th ACM International Conference on Knowledge Discovery and Data Mining*, pp. 1335-1344. ACM, 2015.
- [4] Chen, Tianqi, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems." *arXiv preprint arXiv:1512.01274* (2015).

- [5] Abadi, Martín, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems." In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 265-283. USENIX Association, 2016.
- [6] Jia, Yangqing, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. "Caffe: Convolutional architecture for fast feature embedding." In Proceedings of the 22nd ACM international conference on Multimedia, pp. 675-678. ACM, 2014.
- [7] Yan, Da, Yuzhen Huang, Miao Liu, Hongzhi Chen, James Cheng, Huanhuan Wu, and Chengcui Zhang. "GraphD: Distributed Vertex-Centric Graph Processing Beyond the Memory Limit." IEEE Transactions on Parallel and Distributed Systems (2017).
- [8] Sun, Peng, Yonggang Wen, and Ta Nguyen Binh Duong Xiao. "GraphH: High Performance Big Graph Analytics in Small Clusters." arXiv preprint arXiv:1705.05595 (2017).
- [9] Rosen, Rami. "Linux containers and the future cloud." Linux J 240, no. 4 (2014): 86-95.
- [10] Che, Jianhua, Congcong Shi, Yong Yu, and Weimin Lin. "A synthetical performance evaluation of openvz, xen and kvm." In Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific, pp. 587-594. IEEE, 2010.
- [11] Ghodsi, Ali, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. "Dominant resource fairness: fair allocation of multiple resource types." In Proceedings of the 8th USENIX conference on Networked systems design and implementation (NSDI), pp. 323-336. USENIX Association, 2011.
- [12] Deb, Kalyanmoy. "Multi-objective optimization." In Search methodologies, pp. 403-449. Springer US, 2014.
- [13] Grandl, Robert, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram

- Rao, and Aditya Akella. "Multi-resource packing for cluster schedulers." In ACM SIGCOMM Computer Communication Review, vol. 44, no. 4, pp. 455-466. ACM, 2014.
- [14] Kellerer, Hans, Ulrich Pferschy, and David Pisinger. "Introduction to NP-Completeness of knapsack problems." In Knapsack problems, pp. 483-493. Springer Berlin Heidelberg, 2004.
- [15] Albert, Shon. "Solving mixed integer linear programs using branch and cut algorithm." Masters of Mathematics, NCSU, Fall (2006).

Chapter 4

MetaFlow: A Scalable Metadata Lookup Service

In large-scale distributed file systems, efficient metadata operations are critical, since most file operations have to interact with metadata servers first. In existing distributed hash table (DHT) based metadata management systems, the lookup service could be a performance bottleneck due to its significant CPU overhead. In this work, we present MetaFlow, a scalable metadata lookup service utilizing the software-defined networking (SDN) technique to distribute lookup workload over network components. MetaFlow tackles the lookup bottleneck problem by leveraging B-tree, which is constructed over the physical topology of a data center, to generate flow tables for SDN-enabled switches. Thus, metadata requests could be forwarded to appropriate metadata servers using only network switches. Extensive performance evaluations showed that MetaFlow increases the system throughput by a factor of up to 6.5, reduce system latency by a factor of up to 5 for the metadata management, and speed up machine learning applications by a factor of up to 1.81, compared to existing DHT-based approaches.

4.1 Introduction

Metadata is “data about data”. It summarizes the basic information regarding files and directories in file systems. Normally, a metadata object is a key-value pair, where the key denotes the file name, and the value consists of a set of attributes (e.g., file size, permission, access time, disk block) for the file or directory. In file systems, before users could perform file-related operations such as open, read, write, delete, etc., they have to acquire the files’ metadata first.

Distributed file systems like HDFS, GFS, Lustre and Ceph usually separate the management of metadata from storage servers. Such separation could make it easier to scale the storage capacity and bandwidth of the file system, since new storage servers can be added to the cluster when needed. In these systems, clients should interact with the metadata server first to fetch files’ addresses and other attributes, after that they could perform operations on the desired files. It has been shown in previous work, e.g., [1], [2], that more than 80% of file operations need to interact with metadata servers. Therefore, efficient metadata operations are critical for distributed file systems’ performance. In [3], the authors showed that an optimized metadata management system could improve the performance of directory operations in Lustre significantly (more than 20 times).

The traditional single metadata server scheme cannot cope with the increasing workload in large-scale storage systems. Therefore, most modern distributed file systems like GFS, Ceph, and Lustre deploy a cluster to share the metadata workload. In these systems, a very large metadata table is partitioned into smaller parts located on separate metadata servers. It might be straightforward to achieve large-scale metadata storage by just adding more servers; however, the same cannot be said for improving the system performance. Many novel distributed metadata management systems have been proposed to provide high-performance metadata services, for instance [3], [4], [5], [6], [7].

Figure 4.1 Existing distributed metadata management system architecture. Each node has two subsystems to process lookup requests and I/O requests separately. An I/O operation must wait for the completion of its associated lookup operation.

Existing approaches focused on building overlay-based distributed metadata management systems, as shown in Figure 4.1. These systems usually provide two main operations: lookup and I/O. The lookup operation aims to locate the desired metadata, and the I/O operation retrieves the metadata itself from the storage server using the address returned by the lookup operation. As shown in Chapter 2, one solution is to use a **Central Coordinator** to locate metadata objects. To avoid the single-node failure issue, existing distributed metadata management systems usually use **Distributed Hash Table (DHT)** for metadata lookup. With DHT, a lookup request is processed by several nodes, each of which maintains partial storage information for other nodes. Two DHT models are widely used:

- **Chord** [8]. In a N -node system, each Chord node maintains $O(\log N)$ other nodes' storage information. On average, each lookup request needs to interact with $O(\log N)$ nodes to locate a metadata object.
- **One-Hop** [9]. One-Hop allows each node to maintain all other nodes' information. Any lookup requests will be processed by only one node.

However, the throughput and latency of these DHT-based approaches could be

significantly degraded due to the bottleneck created by a large number of lookup operations. Such operations compete for CPU cycles with I/O operations, which might lead to reduced system throughput. In addition, it could take a long time to locate a metadata object in DHT-based systems, which might increase the system latency for the distributed metadata management.

In this work, we propose MetaFlow, a new efficient and fast distributed lookup service for metadata management. Rather than setting up a separate lookup operation, MetaFlow utilizes network components to locate the desired metadata with two techniques: software-defined-networking (SDN) and a B-tree based overlay network. SDN provides the ability for network switches to forward packets based on metadata identifiers. A B-tree based overlay is constructed over all physical switches and servers in the data center to generate and maintain forwarding tables for SDN-enabled switches. In this way, a metadata request can reach its target server directly, without the need for a separate lookup operation to query the destination. As a result, the latency in metadata operations could be reduced; and more available CPU cycles would be dedicated to I/O operations to improve the system throughput.

The rest of this chapter is structured as follows. Section 4.2 gives a measurement study on existing DHT-based metadata management systems. Section 4.3 shows MetaFlow system design. Section 4.4 gives algorithms using B-tree to generate and maintain flow tables for SDN-enabled switches. The evaluation results are detailed in Section 4.5 and Section 4.6 concludes this chapter.

4.2 Identify Bottlenecks in DHT-based Systems

We construct a testbed to carry out a series of experiments to identify potential performance issues in DHT-based distributed metadata management systems.

4.2.1 Experiment Configurations

We conduct experiments using a testbed with up to 200 VMs and 3 physical switches. In the testbed, we implement two DHT-based models (i.e., Chord and One-Hop), and one centralized model (i.e., Central Coordinator) to provide the lookup service. For the throughput measurement, we compare these systems to an ideal system, which provides linear throughput performance. For the latency measurement, we use the hash-based mapping approach, which has no additional lookup latency, as the baseline. In the experiments, each metadata object is 250 and 290 bytes of a key-value pair, respectively. We set up a set of clients to generate a mix get and put metadata operations with a ratio of 20% and 80%, respectively, to simulate the real metadata workload [10]. In the get operation, a client retrieves a metadata object using the given file name. In the put operation, a client writes new data into a metadata object. We run experiments using four different storage subsystems:

- **Redis:** Redis is an in-memory key-value storage system. HDFS, GFS and PVFS only use memory to store the metadata.
- **LevelDB (HDD):** LevelDB (HDD) is a fast key-value storage system using both the memory and the hard disk drive (HDD). Some file systems like Tablefs [11] store the metadata using LevelDB.
- **LevelDB (SSD):** LevelDB (SSD) uses both the memory and the solid-state disk (SSD) to manage key-value data.
- **MySQL (HDD):** MySQL is a conventional relational database management system (RDBMS). We deploy MySQL on the HDD. It serves as a lower bound for storage subsystems' performance in our experiments.

4.2.2 Throughput

Figure 4.2 shows that DHT could cause large throughput reductions, where the throughput is defined as the maximum number of metadata operations that a

Figure 4.2 Metadata operation throughput comparison between three overlay-based metadata management systems (Control Coordinator, Chord and One-Hop) in the testbed with four storage subsystems: MySQL, LevelDB (HDD), LevelDB (SSD), and Redis.

metadata cluster can handle. DHT solves the single node performance bottleneck problem in Central Coordinator, and could provide higher throughput. However, compared to the ideal system, Chord has roughly 70% throughput reduction with 200 Redis servers as shown in Figure 4.2 (a). The corresponding measure for One-Hop is 50%. When using LevelDB (HDD) and LevelDB (SSD), DHT-based systems still have no less than 20% throughput reduction as shown in Figure 4.2 (b) (c). Even when we use a storage subsystem with low I/O throughput like MySQL (Figure 4.2 (a)), there is still roughly 10% performance reduction for DHT-based approaches with 200 servers.

We carry out system profiling to identify the source of significant throughput reduction in DHT-based systems with LevelDB and Redis. Figure 4.3 shows that such reductions are mainly caused by high CPU cycle consumption of the lookup

Figure 4.3 The lookup subsystem’s CPU overhead in Chord and One-Hop based metadata management systems using four types of storage subsystems: MySQL (HDD), LevelDB (HDD), LevelDB (SSD), and Redis.

service. In the Chord-based system, the lookup subsystem consumes about 70% of CPU cycles with more than 100 Redis nodes. The corresponding measures for LevelDB (HDD) and LevelDB (SSD) are 55% and 60%, respectively. As a result, the storage subsystem might not have enough CPU resource to deal with I/O operations. Although the One-Hop based system has better performance, its lookup service still consumes roughly 40%, 35%, and 25% of CPU cycles when using Redis, LevelDB (SSD), and LevelDB (HDD) as the storage subsystems with more than 100 servers, respectively.

In summary, the lookup service could reduce the system throughput significantly in DHT-based systems, especially when used with in-memory storage subsystems like Redis, because of the CPU resource competition:

- Each lookup request consumes a large portion of CPU cycles compared with I/O requests.
- The lookup subsystem deals with a huge number of lookup requests. Based on Chord’s properties, the Chord-based system should process $\log_2 M$ times more lookup requests than I/O requests on average in a M -node cluster. The One-Hop-based system performs the same number of lookup operations with I/O operations.

Figure 4.4 Metadata operation latency comparison between three overlay-based metadata management systems (Control Coordinator, Chord and One-Hop) in the testbed with four storage subsystems: MySQL, LevelDB (HDD), LevelDB (SSD), and Redis.

4.2.3 Latency

Figure 4.4 shows that DHT-based systems have high system latency, which is defined as the time used to complete a get or put metadata operation. In particular, the Chord-based system is about 8 times slower than the hash-based system when using Redis as the storage subsystem as shown in Figure 4.4 (d). The One-Hop-based system is much faster than the Chord-based system, but it is still 2 times slower than the hash-based system. When using LevelDB (HDD) and LevelDB (SSD) as the storage subsystems, Chord and One-Hop also have obviously higher system latency than the hash-based system by a factor of at least 1.8 and 1.3, respectively, as shown in Figure 4.4 (b) and (c). Figure 4.4 (a) shows that DHT-based systems perform acceptably when used with MySQL. However, compared to other systems, storage systems using MySQL have much higher latency.

Figure 4.5 The lookup subsystem’s latency overhead in Chord and One-Hop based metadata management systems using four types of storage subsystems: MySQL (HDD), LevelDB (HDD), LevelDB (SSD), and Redis.

Profiling results in Figure 4.5 show that the high system latency in DHT-based systems is mainly caused by the lookup operation. In our experiments, the lookup operation in the Chord-based system could account for 72% to 84% of the total system latency when using Redis as the storage subsystem. Although the One-Hop-based system has better performance, its lookup operation still takes roughly 35% of the total system latency. When using LevelDB (HDD) as the storage subsystem, the lookup operation takes at least 40% and 10% of the total system latency for the Chord-based system and the One-Hop-based system, respectively. In summary, the lookup service in DHT-based systems adds considerable latency to metadata operations. The main reason is that a lookup operation needs to invoke at least one remote procedure call (RPC) on storage servers. Based on Chord’s properties, the Chord-based system invokes $\log_2 M$ RPCs on average to locate a metadata object in a M -node cluster. The One-Hop-based system needs to one RPC per metadata operation.

4.3 MetaFlow System Design

We propose MetaFlow to solve the performance bottleneck caused by lookup operations in existing DHT-based approaches for the metadata management. This section describes the design objective, architecture, and the lookup workflow.

4.3.1 Design Objective

We design MetaFlow to provide a fast lookup service with minimal overhead incurred on metadata storage servers. Essentially, MetaFlow maps a MetaDataID, which is the hash value of a metadata object's file name, to the location of a server storing this metadata object. Compared to DHT-based approaches, MetaFlow has two key features:

- **In-Network Lookup.** MetaFlow places the lookup workload on network components instead of metadata storage servers. It takes advantage of SDN-enabled switches to send metadata requests directly to storage servers using just MetaDataIDs instead of IP or MAC addresses. As a result, MetaFlow could avoid the CPU resource competition problem between the lookup subsystem and the storage subsystem in conventional DHT-based approaches.
- **Zero-Hop.** MetaFlow does not have a separate step to fetch the location of the desired metadata object. It allows the client to directly establish a network connection to a metadata storage server to perform I/O operations like *get*, *put*, *update*, *delete* with MetaDataIDs. In the following example, we illustrate how this could be done using an HTTP connection:

```
MetaDataID=Hash(FileName)
Connection=HTTPConnection(MetaDataID, Port)
// Method denotes metadata operation, such as Get, Put, Update, Delete.
Connection.request(Method)
Connection.getResponse()
Connection.close()
```

4.3.2 System Architecture

MetaFlow has four key components: Storage Cluster, Application Cluster, SDN-based Networking, and MetaFlow Controller.

Application Cluster. The application cluster manages a set of clients, which query metadata objects for distributed file system related operations.

83

rules, SDN decouples the control and data planes of a network. A centralized SDN controller manages the rules for the switch to forward packets.

MetaFlow uses OpenFlow as the standard for SDN. There are three components in an OpenFlow architecture: an OpenFlow-enabled switch, which uses flow entries to forward packets; an OpenFlow controller, which manages flow tables; and a secure channel, which connects the controller to all switches. A packet is examined with regard to the flow entries by using one or more its header fields. If there is a match, the packet is processed according to the instruction in the flow entry. If not, the packet is sent to the OpenFlow controller for further processing. OpenFlow 1.0.0 is one of the most widely used specifications. It supports 12 header fields, which include Source/Destination IP Address, Source/Destination TCP/UDP Port, etc. MetaFlow uses destination IP address and destination TCP port to forward packets.

MetaFlow packets are normal IP packets. MetaFlow Request is the packet sent by clients to query a metadata object with the MetaDataID, and MetaFlow Response is the packet sent from storage servers with the desired metadata object as the packet's content. MetaFlow packets differ from common HTTP packets in two aspects: destination IP address and destination TCP port.

- **Destination IP Address.** MetaFlow uses the MetaDataID, which is the hash value of a metadata object's file name, as the MetaFlow Request's destination IP address. Depending on the IP protocol in use, MetaDataID could have a different length. MetaDataID is a 32-bit integer when using IPv4. In IPv6, MetaDataID is a 128-bit integer.
- **Destination TCP Port.** MetaFlow uses the destination TCP port of the IP packet to distinguish MetaFlow packets from other packets in the cluster. The MetaFlow Request uses 9000 as its destination TCP port in Figure 4.6. SDN-enabled switches could detect MetaFlow Requests based on the destination TCP port, and process them using the appropriate flow tables. Other packets

will be relayed using existing layer 2/3 switching techniques. We should note that normal network packets except MetaFlow packets should not use 9000 as their destination TCP ports in Figure 4.6.

MetaFlow Controller. The MetaFlow Controller is in charge of generating and maintaining flow tables for SDN switches to forward metadata packets.

4.3.3 MetaFlow Packets Forwarding

MetaFlow uses SDN-enabled switches to forward MetaFlow packets properly. More specifically, it relays MetaFlow Requests to corresponding storage servers based on MetaDataIDs. It also forwards MetaFlow Responses back to clients.

Forward MetaFlow Requests. As shown in Figure 4.6, MetaFlow forwards a MetaFlow Request packet via three steps:

- **From the application cluster to the storage cluster.** The SDN-enabled switch in the application cluster forwards the MetaFlow Request based on the destination TCP port. As shown in Figure 4.6, SwitchD is configured to forward the MetaFlow Request, whose destination TCP port is 9000, to SwitchA, which is located in the storage cluster.
- **From the storage cluster to the storage server.** The SDN-enabled switch in the storage cluster forwards MetaFlow Requests based on both the destination TCP port and the destination IP address. As shown in Figure 4.6, when receiving packets from SwitchA, SwitchB recognizes the MetaFlow Request based on the destination TCP port, and compares its MetaDataID against its flow table based on a longest prefix match algorithm. According to the instruction from the matched flow entry, SwitchB forwards the MetaFlow Request to ServerA.
- **From the network layer to the application layer in the metadata storage server.** The server in the storage cluster forwards the MetaFlow Request to

the application layer from the network layer. Normally, the storage server will drop the received MetaFlow Request, since its destination address is the MetaDataID rather than the server's IP address. To address this problem, MetaFlow deploys a network address translation (NAT) agent on each storage server to replace the MetaFlow Request's destination address with its IP address. Therefore, the application layer in the storage server could receive and process MetaFlow Requests.

Forward MetaFlow Responses. MetaFlow Responses are relayed back to clients based on clients' physical IP addresses. Normally, the storage server will put its physical IP address in the MetaFlow Response's source address field. However, the client will drop these MetaFlow Responses, since the requests' destination address differ from the responses' source address. For example, a client sends out a request using the MetaDataID 155.69.146.43 as the destination IP address. The corresponding storage server sends back the response using its physical IP address 192.168.0.1 as the source IP address. In this case, the client will drop the response, since it expects a response from 155.69.146.43 rather than 192.168.0.1. To solve this problem, MetaFlow uses the storage server's NAT agent to replace the MetaFlow Response's source address field with the original MetaDataID before sending out the response.

4.4 Flow Table Management

The central problem in implementing MetaFlow is how to manage flow tables for the SDN-enabled switches in the storage cluster. Since MetaFlow places the lookup workload on network components, we have to generate flow tables for the SDN-enabled switches in both the application cluster and the storage cluster. Flow tables for the application cluster's switches could be generated easily, since these switches forward all MetaFlow Requests to a pre-determined destination, which is the storage cluster. However, in the storage cluster, the MetaFlow

Requests' destination storage servers are not known in advance. Therefore, existing IP-based routing protocols, which include both conventional routing protocols (e.g., RIP, OSPF), and centralized SDN-based routing protocols cannot work. In this section, we describe the flow table generation and management algorithm in MetaFlow. The algorithm has two components:

- **Flow Table Generation.** The MetaFlow Controller distributes all metadata objects across the storage cluster and generates corresponding flow tables for the SDN-enabled switches in the storage cluster.
- **Flow Table Maintenance.** When a storage node joins or leaves the storage cluster, the MetaFlow Controller updates appropriate flow tables for the SDN-enabled switches in the storage cluster.

4.4.1 Flow Table Generation

MetaFlow uses a B-tree to distribute metadata objects among the storage servers, and generates corresponding flow tables for SDN-enabled switches. First, the MetaFlow Controller maps the storage cluster's network topology, which could be a tier tree [12] or fat tree [13], to a logical B-tree data structure. Using the B-tree's property, MetaFlow then distributes the metadata objects across storage servers, and generates appropriate flow tables for SDN-enabled switches.

Tree Topology. MetaFlow has to be able to work with different tree topologies in data centers. There are two widely used tree topologies: tier tree and fat tree.

- A tier tree network consists of two or three layers of network switches. A three-tier tree network contains an edge layer, connecting servers via top of rack (ToR) switches; an aggregation layer, using end of rack (EoR) switches to connect ToR switches; and a core layer at the root of the tree. There is no aggregation layer in a two-tier tree network.
- A fat tree network is an extended version of the three-tier tree network. Pod is the basic cell of a fat tree network. Assume that each switch has n switch

Figure 4.7 A B-tree example. In this B-tree, there are 2 inner nodes and 6 leaf nodes. Each node can store 2 keys at most.

ports in a fat tree network, a Pod consists of $n/2$ aggregation layer switches, $n/2$ edge layer switches, and their connected servers. Therefore, each edge layer switch connects $n/2$ aggregation layer switches, and each core layer switch connects $n/2$ core switches in the fat tree network.

B-Tree. B-tree is a self-balancing tree data structure, which has two key features: 1) it distributes the key-value data across its nodes in a balanced manner, 2) it allows lookup operations in logarithmic time. A B-tree is made up of three types of nodes: leaf node, inner node and root node. We adopt the widely-accepted definition by Comer [14], where these nodes construct a sample B-tree as shown in Figure 4.7.

A B-tree stores key-value pairs in its nodes in non-decreasing order of the keys' values. The key-value pair stored in the non-leaf node also acts as the partition value to separate the subtree. For example, in Figure 4.7, the root node partitions the B-tree into two subtrees. In particular, the left subtree stores the keys less than 100, and the right subtree manages the keys larger than 100.

Each B-tree node has the same storage capacity. For example, we set each node can store $2d - 1$ keys at most, where d is an integer. If a node manages more than $2d - 1$ keys, it will be divided into two nodes, each of which stores d keys.

Figure 4.8 Mapping a three-tier tree network to a logical B-tree. The core layer switch is mapped to the B-tree’s root node; the aggregation and edge layer switches are mapped to the inner nodes; and the storage servers are mapped to the leaf nodes.

Meanwhile, the partition value of the $2d - 1$ keys is inserted into the node’s parent. If there is no parent (i.e., the node is the root node), a new root node will be created for the B-tree. Therefore, except the root node, all the nodes in B-tree manages at least $d - 1$ keys and at most $2d - 1$ keys. The root node stores at least 1 keys and at most $2d - 1$ keys.

To search a key, the B-tree is recursively traversed from top to bottom starting at the root node. At each level, the search algorithm chooses the subtree according to the comparison result between the desired key and stored partition values. For example, in Figure 4.7, if a client queries the key 65, it will choose the left subtree at the root node. Then it chooses the middle subtree at the inner node, since $65 > 30$ and $65 < 80$. Finally, the client can fetch the value of key 65 at the correct leaf node.

Mapping Physical Network Topology to Logical B-tree. MetaFlow uses a logical B-tree to manage the storage cluster. To do that, MetaFlow first discovers the physical storage cluster’s topology through OpenFlow, and then carries out a

Figure 4.9 Mapping a fat tree network to a logical B-tree. In this network, each switch has four switch ports. Two aggregation layer switches, two edge layer switches, and four storage servers form a Pod in the fat tree network. To map the fat tree network to a B-tree, the four core layer switches are mapped to one B-tree’s root node. The two aggregation layer switches in the same Pod are mapped to one inner node. Each edge layer switch and storage server is mapped to an inner node and a leaf node, respectively.

mapping operation from the physical topology, which includes storage servers, SDN-enabled switches, and network links, to a logical B-tree. This is done in the MetaFlow Controller via the following steps:

- Storage servers are mapped to the B-tree’s leaf nodes.
- The core switch is mapped to the B-tree’s root node.
- The aggregation and edge switches are mapped to the inner nodes in the B-tree based on the layers that they are in.
- Network links are mapped to the logical connections between parent nodes and child nodes in the B-tree.

This mapping strategy works for both the tier tree and fat tree network. It is straightforward to map a tier tree network to a B-tree as shown in Figure 4.8, since they are quite similar in terms of structure. To map a fat tree network to a

B-tree, MetaFlow might need to map multiple switches to one B-tree node. For example, in Figure 4.9, a pod contains 2 aggregation layer switches, 2 edge layer switches and 4 storage servers. To map this fat tree to a B-tree, the 4 core layer switches are mapped to one B-tree root node. The 2 aggregation layer switches in the same Pod are mapped to one inner node. Edge layer switches and storage servers are mapped to inner nodes and leaf nodes, respectively.

MetaFlow uses two states, which are *idle* and *busy* states, to simulate the B-tree node creation operation. The classical B-tree can create new nodes dynamically for node split operations. MetaFlow uses *idle* and *busy* states to simulate this operation. In the idle state, the physical node, which can be a storage server or an SDN-enabled switch, contains no data. The busy state means that the physical node manages some keys. For example, in Figure 4.9, there is no data stored in the storage cluster initially. In this case, all the nodes are in the *idle* state. When some key-value pairs are inserted, some nodes' states are transformed into *busy*. When a node is full, the mapped B-tree activates an *idle* node to store roughly half of the full node's data.

Generating Flow Table. MetaFlow transforms the B-tree's partition values to SDN-enabled switches' flow tables. The key challenge is how to represent the partition values in a format that can be recognized by SDN-enabled switches. Since current OpenFlow-enabled switches only support longest prefix matching algorithm to deal with the destination IP address field, MetaFlow uses Classless Inter-Domain Routing (CIDR) blocks to represent partition values in the B-tree.

A CIDR block is a group of IP addresses with the same routing prefix. For example, the IPv4 CIDR block 155.69.146.0/24 represents 256 IPv4 addresses from 155.69.146.0 to 155.69.146.255. An OpenFlow-enabled switch can use a CIDR block as its forwarding table entry. For example, the entry

“155.69.146.0/24 → 192.168.0.1”

Figure 4.10 Using CIDR blocks to generate flow tables in a logical B-tree.

means that a packet will be forwarded to 192.168.0.1 if its destination IP address ranges from 155.69.146.0 to 155.69.146.255.

MetaFlow uses CIDR blocks to map the B-tree's partition values to SDN-enabled switches' flow tables. In Figure 4.10, SwitchC splits the set of all metadata objects into two partitions, which are 0.0.0.0/1 and 128.0.0.0/1, using 128.0.0.0 as the partition value. In this case, SwitchA is responsible for metadata objects with MetaDataIDs that are less than 128.0.0.0. SwitchB is responsible for metadata objects with MetaDataIDs that are no less than 128.0.0.0. Therefore, MetaFlow generates the following flow table for SwitchC:

Destination Address	Destination TCP Port	Action
0.0.0.0 / 1	9000	Forward to SwitchA
128.0.0.0 / 1	9000	Forward to SwitchB

Normally, a B-tree partition value could generate a list of flow entries. As shown in Figure 10, SwitchA splits the allocated metadata objects using 96.0.0.0 as the partition value. In this case, MetaFlow uses block 0.0.0.0/2 and block 64.0.0.0/3 to represent the left partition of SwitchA. Block 96.0.0.0/3 represents the right partition of SwitchA. MetaFlow then generates three flow entries for SwitchA to represent the partition value 96.0.0.0 in the B-tree:

Destination Address	Destination TCP Port	Action
0.0.0.0 / 2	9000	Forward to Server1
64.0.0.0 / 3	9000	Forward to Server1
96.0.0.0 / 3	9000	Forward to Server2

4.4.2 Flow Table Maintenance

In a dynamic network, storage servers and switches can join and leave the system anytime. MetaFlow needs to update appropriate flow tables in the SDN-enabled switches to maintain proper lookup operations.

Node Joins. When a new storage server or a switch is added to the storage cluster, MetaFlow creates a new node in the existing B-tree at the appropriate location, and sets its state to *idle*. Initially, the new node will not be allocated with metadata objects immediately when it joins the system. Therefore, there is no change to the existing flow tables.

Node Leaves. If a storage server fails, the corresponding B-tree node will be deleted. In this case, MetaFlow activates an *idle* node having the same parent node to replace the failed node. MetaFlow identifies the parent switch of the failed node and the newly activated node. Then, it updates appropriate flow entries in the parent switch, using the newly activated node to replace the failed node. If there is no available *idle* node to handle the failed node, more storage servers should be added to the cluster to meet the storage requirement.

Node Splits. A B-tree node would be split into two nodes when it is full. In this case, the mapped B-tree of MetaFlow activates an *idle* node into the *busy* state, and transfers part of the metadata objects from the full node to the newly activated node. Finally, MetaFlow updates appropriate flow entries to maintain proper lookup operations. In Figure 4.11, ServerA is split into two nodes using 80.0.0.0 as the partition value. Before the split operation, SwitchD has the following flow table for the left subtree:

Figure 4.11 Splitting a B-tree node. In this example, when ServerA is full, the MetaFlow controller activates ServerC, uses 80.0.0.0 to split ServerA, and transfers the CIDR block 80.0.0.0/4 from ServerA to ServerC.

Destination Address	Destination TCP Port	Action
0.0.0.0 / 2	9000	Forward to ServerA
64.0.0.0 / 3	9000	Forward to ServerA

After the split, one flow entry is added, and another one is modified:

Destination Address	Destination TCP Port	Action
0.0.0.0 / 2	9000	Forward to ServerA
64.0.0.0 / 4	9000	Forward to ServerA
80.0.0.0 / 4	9000	Forward to ServerC

The main problem is how to split the logical B-tree node. In a classical B-tree, we can just select the middle value to partition the B-tree node. However, for the logical B-tree in MetaFlow, we have to select a partition value that is recognizable by network switches. MetaFlow uses a traversal algorithm to solve this problem. Given a full B-tree node, which is allocated with several ordered CIDR blocks of metadata objects, MetaFlow splits it to two sets: a left set and a right set. The left set will be left in the existing node, and the right set will be transferred to the newly activated node. MetaFlow splits a full node via following steps:

1) MetaFlow traverses the ordered CIDR blocks in the full node, and checks the number of metadata objects for each block.

2) During the traversal operation, MetaFlow puts the incoming CIDR block into the left set, until the left set's number of metadata objects exceeds 40% of the full node's number of metadata objects. There are two cases:

- **The number of metadata objects in the left set is smaller than 60% of that in the full node.** In this case, MetaFlow puts the rest of CIDR blocks into the right set. For example, a full node contains three ordered CIDR blocks: 192.168.100.0/25, 192.168.100.128/25, and 192.168.100.192/26. When putting 192.168.100.0/25 into the left set, if the left set's number of metadata objects is between 40% and 60% of the original node's number of metadata objects, MetaFlow puts the rest of CIDR blocks, which are 192.168.100.128/25 and 192.168.100.192/26, into the right set.
- **The number of metadata objects in the left set is more than 60% of that in the full node.** In this case, MetaFlow would remove the most recently considered CIDR block from the left set. This CIDR block will be evenly split into two sub-blocks to replace the original one. For example, the CIDR block 192.168.100.0/24 will be split into two sub-blocks: 192.168.100.0/25 and 192.168.100.128/25. MetaFlow uses these two sub-blocks to replace the original CIDR block, and continues the traversal operation. Normally, MetaFlow will then process the CIDR block 192.168.100.0/25 using the same operation in Step 2.

3) After the traversal operation, MetaFlow transfers the CIDR blocks in the right set to the newly activated node, and updates the appropriate flow entries.

After these three operations, the flow table on the affected switch converge, and would not change until a new node split operation is required.

It is essential to reduce the number of flow entries generated by the node split operation, since most SDN-enabled switches can only support a few thousands of flow entries. If the right set contains exactly 50% of the full node's metadata

objects, our experiments show that the node split operation usually ends after a few tens of iterations in Step 2, where each iteration generates a new flow entry for its parent switch. In real-world systems, the growing size of flow table can significantly limit the system performance and scalability. When we use a value between 40% to 60%, we find that Step 2 can end after just several iterations. Compared to the value of 50%, a value between 40% to 60% can reduce the number of new flow entries by a factor of up to 10. As a trade-off between storage balance and flow table size, MetaFlow uses a value between 40% to 60% for the node split operation.

4.5 Performance Evaluations

In this section, we evaluate MetaFlow performance using both a large-scale simulation and a testbed with 200 VMs. Specifically, we investigate MetaFlow’s performance in terms of throughput and latency using extensive experiments with realistic metadata workload models.

4.5.1 Experiments’ Parameters and Configurations

Performance Measures. We evaluate the performance of a MetaFlow-based metadata management system, and compare it to existing DHT-based approaches like Chord and One-Hop.

- *Throughput.* We measure the system throughput with increasing cluster size in both the simulator and the testbed. In the experiments, we define system throughput as the maximum number of metadata operations that a metadata cluster can deal with.
- *Latency.* We measure the system latency with increasing cluster size in both the simulator and the testbed. In the experiments, we define system latency as the average time used to complete a metadata operation.

System Configurations. We implement a distributed metadata management system using MetaFlow in both a simulator and a testbed.

Simulation Setup. The simulator has up to 2000 servers, forming a storage cluster based on the fat tree topology. In the cluster, each switch has 32 ports. Thus, 16 edge layer switches, 16 aggregation layer switches, and 256 servers form a pod in the fat tree. There are 32 core layer switches in total. All network connections between switches and servers are 10-Gbps links. The simulator uses 500 clients to form an application cluster for generating metadata workload.

Testbed Setup. The testbed has 200 VMs, forming a storage cluster based on a two-tier tree topology. In the testbed, we use three Extreme Summit x670c switches with OpenFlow 1.0.0 support. Each edge layer switch connects up to 100 VMs, each of which is allocated with a 2 GHz CPU core and 4 GB memory. All the network connections between switches and servers are 10-Gbps links. The testbed uses 50 clients as an application cluster for generating metadata workload. Two VMs are set up to manage the switches' flow tables. One is the OpenFlow Controller, which manages flow tables for switches using OpenFlow protocols. The other one is the MetaFlow Controller, which generates and maintains flow tables for lookup operations using B-tree.

Workloads. We use a metadata workload in which 80% are *get* operations and 20% are *put* operations. This is similar to real-world metadata workloads [10]. In the *get* operation, a client retrieves a metadata object using the given MetaDataID. In the *put* operation, a client writes new data into a metadata object. Each metadata object for a file and directory is a key-value pair with the size of 250 and 290 bytes, respectively. This is similar to the metadata object size in HDFS.

Experiment Configurations. We use different types of storage subsystems in the experiments to measure the system throughput and latency.

- **Testbed.** We test the MetaFlow-based system's throughput and latency using Redis, LevelDB (SSD), LevelDB (HDD), and MySQL as the storage.
- **Simulator.** We conduct several tests to find the appropriate throughput and latency parameters to be used in the simulations. In these tests, we measure the performance of a lookup subsystem and four different storage subsystems, namely Redis, LevelDB (SSD), LevelDB (HDD), and MySQL, on a single CPU core. Based on these results, we define a throughput and a latency ratio to be used in the simulations. More specifically, the throughput ratio is obtained by dividing the throughput of the lookup subsystem to that of a storage subsystem. Similarly, the latency ratio is obtained by dividing the latency of the lookup subsystem to that of a storage subsystem. Therefore, in the simulations, we use the following throughput ratios: 1, 1.5, 2, and 100 to reflect the throughput performance of Redis, LevelDB (SSD), LevelDB (HDD), and MySQL, respectively. Similarly, the following latency ratios are used: 1, 0.7, 0.5, and 0.001. These ratios reflect the latency performance of Redis, LevelDB (SSD), LevelDB (HDD), and MySQL, respectively.

4.5.2 Throughput Results

Figure 4.12 shows the system throughputs in simulations. In these experiments, we compare the throughputs of the MetaFlow-based system and DHT-based systems to those of an ideal system, which has linear performance (i.e., the ideal throughput increases linearly with respect to the cluster size). We observe that MetaFlow consistently performs better than Chord and One-Hop in all situations. When the throughput ratio is 1 (which means the metadata management system has similar I/O and lookup throughputs), MetaFlow has a throughput reduction of 12% to 20% compared to the ideal system. In contrast, Chord and One-Hop have 80% to 85% and 45% to 50% throughput reduction, respectively. When using other throughput ratios such as 2, MetaFlow has about 12% to 17% throughput reduction. The corresponding measures for Chord and One-Hop are

Figure 4.12 Throughput comparison in the simulation between the MetaFlow-based system and two DHT-based systems (Chord and One-Hop) with four lookup/storage throughput ratios: 100, 2, 1.5, and 1.

75% to 80% and 30% to 36%, respectively. Even in a low-throughput storage system such as MySQL, MetaFlow is still better, but not by much as shown in Figure 4.12 (a). The reason is that the limiting factor in MySQL-based systems is Disk I/O throughput, not lookup. It should be noted that such low-throughput systems are not suitable for large-scale metadata management in practice. The results for MySQL provided here are mainly for highlighting the lower bound of MetaFlow's performance.

Results from the testbed confirm performance improvement demonstrated in the simulations. In Figure 4.13 (d), we observe that MetaFlow has roughly 15% of throughput reduction compared to the ideal system when using 200 Redis servers. On the contrary, Chord and One-Hop have throughput reductions of nearly 70% and 45%, respectively. As shown in Figure 4.13 (b) and (c), MetaFlow has about 8% and 15% throughput reduction for LevelDB (HDD) and LevelDB (SSD),

Figure 4.13 Throughput comparison between the MetaFlow-based system and two DHT-based systems (Chord and One-Hop) using the testbed with four types of storage subsystems: MySQL, LevelDB (HDD), LevelDB (SSD), and Redis.

respectively. In contrast, One-Hop suffers roughly 20% and 40% performance reduction. At the same time, Chord has even more performance reduction, about 50% and 65%.

4.5.3 Latency Results

In this set of experiments, we compare the request latency of the MetaFlow-based system and other DHT-based systems with respect to a hash-based system, which has no metadata lookup latency. Figure 4.14 shows that the MetaFlow-based system consistently has lower latency than Chord and One-Hop in the simulations. In particular, when the latency ratio is 1, MetaFlow's latency is about up to 1.5 times higher than the hash-based system. In contrast, Chord and One-Hop's latency are roughly 7 and 2 times higher than the hash-based system.

Figure 4.14 Latency comparison in the simulation between the MetaFlow-based system and two DHT-based systems (Chord and One-Hop) with four lookup/storage latency ratios: 0.001, 0.5, 0.7, and 1.

When using other lookup/storage latency ratios like 0.7 and 0.5, MetaFlow has up to 5% more latency than the hash-based system. At the same time, Chord and One-Hop have at least 50% and 20% more latency using the same setting, respectively as shown in Figure 4.14 (b) (c). In the system with lookup/storage latency ratio of 0.001, as shown in Figure 4.14 (a), MetaFlow and One-Hop have nearly the same latency performance with the hash-based system. The reason is that the high I/O latency in such systems renders lookup latency insignificant. However, we should note that such high latency systems are not suitable for metadata management in distributed file systems.

Testbed results are similar to simulation results with regard to the system latency. Figure 4.15 (d) shows that MetaFlow's latency is roughly 1.6 times higher than the hash-based system when using 200 Redis servers. On the contrary, Chord and

Figure 4.15 Latency comparison between the MetaFlow-based system and two DHT-based systems (Chord and One-Hop) using the testbed with four storage subsystems: MySQL, LevelDB (HDD), LevelDB (SSD), and Redis.

One-Hop suffer up to 6.7 and 2.1 times more latency than the hash-based system, respectively. If the storage subsystem is LevelDB (SSD), MetaFlow has similar latency performance with the hash-based system. Meanwhile, Chord and One-Hop have latencies that are up to 5.5 and 1.6 times higher than the hash-based system. When the metadata management system is deployed over an HDD-based storage system like LevelDB(HDD) and MySQL, MetaFlow and One-Hop have similar latency with the hash-based system as shown in Figure 4.15 (a) and (b). This is mainly because I/O operations on HDD take a large part of the latency.

4.5.4 Real-world Distributed File System Results

We also investigate the performance of a distributed file system, which uses MetaFlow for managing its metadata. The testbed contains 100 storage servers,

Figure 4.16 Distributed file system's performance comparison between MetaFlow and DHT-based approaches. The system contains 100 storage servers and 10 metadata servers. 50 clients are set up to generate background metadata workload in which 20% are get and 80% are put operations. We measure the time for a client to write 100 GB of files into the distributed file system with different file sizes: 64 KB, 256 KB, 16 MB, and 64 MB.

and 10 metadata servers. We generate background metadata workloads in which 20% are get and 80% are put operations. We measure the completion time to write 100 GB of files. To investigate the impact of file sizes, we run the experiment with 4 different file sizes, which are 64KB, 256KB, 16MB, and 64MB.

The results in Figure 4.16 show that file systems having a lot of small files could benefit greatly from MetaFlow. More specifically, if the file size is 64 KB, the MetaFlow-based distributed file system consistently takes the least time to write 100 GB of files, regardless of the background metadata workload. If we generate 500 thousands of metadata requests per second, MetaFlow-based distributed file system uses roughly 6800 seconds to write all 64 KB files. In contrast, Chord and

Figure 4.17 CDF distribution of input image data size.

One-Hop take 8500 and 7500 seconds, which are roughly 25% and 10% longer than MetaFlow-based file system, respectively.

The reason for the performance improvement is that MetaFlow reduces the metadata operation time. To write a file into a distributed file system, the client must fetch the appropriate metadata first, which include directory permission, file location, etc. For small files, the metadata operation time constitutes a large part of the total file operation time. Thus, there is obvious performance improvement for small file operations. However, if the file size is large such as 16 MB and 64 MB, there are not much difference between MetaFlow and other approaches. As shown in Figure 4.16 (c) and (d), MetaFlow takes roughly 1820 and 1320 seconds, which is similar to Chord and One-Hop. The reason is that data writing operation takes much more time than metadata operation for large files.

4.5.5 Real-world Application Results

Distributed machine learning could benefit from MetaFlow when working with large-scale distributed file systems. In this experiment, we set up 100 storage

servers and 10 metadata servers connected by 10Gbps. The 100 storage servers use Redis to manage small image data in memory, and the 10 metadata servers use Redis to manage all metadata. A NVIDIA Titan XP GPU node is set up to perform inference operations using a deep learning framework, Caffe and two pre-trained models, AlexNet and Inception. During the processing, the GPU node fetches an image from the distributed file system, then uses AlexNet or Inception to predict the category of each image, and process 16 images until all images are processed. We use the validation dataset of ImageNet as input, which contains 50000 images. In Figure 4.17, we show the CPF distribution of input image data size. We observe that 80% of images are less than 170 KB. The average data size of the input images is 130 KB.

In following table, we show average time of the GPU node to process an image.

Model	Metadata Lookup Time (ms)			Image Fetch Time (ms)	Computation Time (ms)
	MetaFlow	One-Hop	Chord		
AlexNet	0.17±0.06	0.51±0.12	0.67±0.14	1.75±0.61	1.51±0.72
Inception	0.15±0.08	0.48±0.23	0.63±0.18	1.69±0.85	3.26±0.56

When using AlexNet and MetaFlow to process images, the GPU node roughly uses 0.17ms to lookup the location of the target image, and uses 1.75ms to fetch the image from a specific storage server, and uses 1.51ms to predict its category. Thus, the GPU node averagely uses 3.43s to process an image. When using Chord and One-Hop, the corresponding processing time of AlexNet are 3.77s and 3.93s, respectively. Thus, MetaFlow could accelerate AlexNet inference by a factor of 1.09 and 1.15, compared to Chord and One-Hop. When using Inception for image classification, MetaFlow can speed up the inference operation by a factor of 1.06 and 1.09, compared to Chord and One-Hop.

4.6 Summary

In this work, we propose a novel lookup service for metadata management in data center. Popular DHT-based systems place the lookup subsystem and storage

subsystem on the same server. These two subsystems may compete for CPU resources, which leads to reduced throughput and high latency. MetaFlow solves this problem by transferring the lookup service to the network layer. MetaFlow implements this approach by mapping a data center's network topology to a logical B-tree. Appropriate flow tables will then be generated and distributed to SDN-enabled switches. Therefore, the switches would have the ability to forward metadata requests using just the MetaDataIDs. Experiments show that MetaFlow could increase the system throughput by a factor of up to 6.5, and reduce the system latency by a factor of up to 5 for the metadata management compared to DHT-based approaches. Machine learning applications could also benefit from reduced metadata lookup overhead. We believe that MetaFlow will be a valuable component in many distributed metadata management systems. In the future, we plan to use it in more real-world applications, and enhance its security.

References

- [1] Carns, Philip, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. "Small-file access in parallel file systems." In *Parallel & Distributed Processing*, 2009. IPDPS 2009. IEEE International Symposium on, pp. 1-11. IEEE, 2009.
- [2] Roselli, Drew S., Jacob R. Lorch, and Thomas E. Anderson. "A Comparison of File System Workloads." In *USENIX annual technical conference, general track*, pp. 41-54. 2000.
- [3] Meshram, Vilobh, Xavier Besseron, Xiangyong Ouyang, Raghunath Rajachandrasekar, Ravi Prakash Darbha, and Dhabaleswar K. Panda. "Can a decentralized metadata service layer benefit parallel filesystems?" In *Cluster Computing (CLUSTER)*, 2011 IEEE International Conference on, pp. 484-493. IEEE, 2011.
- [4] Weil, Sage A., Scott A. Brandt, Ethan L. Miller, Darrell DE Long, and Carlos Maltzahn. "Ceph: A scalable, high-performance distributed file system." In *Proceedings of the 7th symposium on Operating systems*

- design and implementation, pp. 307-320. USENIX Association, 2006.
- [5] Aguilera, Marcos K., Wojciech Golab, and Mehul A. Shah. "A practical scalable distributed b-tree." *Proceedings of the VLDB Endowment* 1, no. 1 (2008): 598-609.
 - [6] Li, Weijia, Wei Xue, Jiwu Shu, and Weimin Zheng. "Dynamic hashing: Adaptive metadata management for petabyte-scale file systems." In *23rd IEEE/14th NASA Goddard Conference on Mass Storage System and Technologies*, pp. 93-98. 2006.
 - [7] Brandt, Scott A., Ethan L. Miller, Darrell DE Long, and Lan Xue. "Efficient metadata management in large distributed storage systems." In *Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pp. 290-298. IEEE, 2003.
 - [8] Stoica, Ion, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. "Chord: A scalable peer-to-peer lookup service for internet applications." *ACM SIGCOMM Computer Communication Review* 31, no. 4 (2001): 149-160.
 - [9] Gupta, Anjali, Barbara Liskov, and Rodrigo Rodrigues. "One Hop Lookups for Peer-to-Peer Overlays." In *HotOS*, pp. 7-12. 2003.
 - [10] Dandong, Wang, and Fang Yu. "Decentralized metadata management research based on distributed hash table technology." In *Industrial Control and Electronics Engineering (ICICEE), 2012 International Conference on*, pp. 1892-1895. IEEE, 2012.
 - [11] Ren, Kai, and Garth A. Gibson. "TABLEFS: Enhancing Metadata Efficiency in the Local File System." In *USENIX Annual Technical Conference*, pp. 145-156. 2013.
 - [12] Bari, Md Faizul, Raouf Boutaba, Rafael Esteves, Lisandro Zambenedetti Granville, Maxim Podlesny, Md Golam Rabbani, Qi Zhang, and Mohamed Faten Zhani. "Data center network virtualization: A survey." *IEEE Communications Surveys & Tutorials* 15, no. 2 (2013):

909-928.

- [13] Al-Fares, Mohammad, Alexander Loukissas, and Amin Vahdat. "A scalable, commodity data center network architecture." In ACM SIGCOMM Computer Communication Review, vol. 38, no. 4, pp. 63-74. ACM, 2008.
- [14] Comer, Douglas. "Ubiquitous B-tree." ACM Computing Surveys (CSUR) 11, no. 2 (1979): 121-137.

Chapter 5

ParameterFlow: Optimizing Network Performance for Large-Scale Distributed Machine Learning

The Parameter Server (PS) framework is widely used for distributed machine learning (ML) by having worker nodes performing data-parallel computation, and having server nodes maintaining globally shared parameters. When training ML models, worker nodes would frequently pull parameters from server nodes and push updates to server nodes, resulting in high communication overhead. In this work, we propose a novel communication layer for the PS framework called Parameter Flow (PF) with two techniques. First, we introduce an update-centric communication (UCC) model for exchanging data between worker/server nodes via two operations: broadcast and push. Second, we develop a dynamic value-bounded filter (DVF) to reduce network traffic by selectively dropping updates before transmission. With these two techniques, PF could significantly reduce network traffic and communication time. Extensive performance evaluations showed that PF could speed up popular distributed ML applications by a factor of up to 4.3, compared to a generic PS-based system.

5.1 Introduction

Machine Learning (ML) builds models from training data, and use them to make predictions on new data. It is used in a wide range of applications, such as image recognition and recommender systems. Typically, an ML model consists of a large number of parameters, represented as vectors and matrices. To minimize the prediction error, an ML application usually uses an iterative-convergent algorithm, such as stochastic gradient descent (SGD), to train a given ML model.

In industrial ML applications, the size of training data sets could be hundreds to thousands of terabytes. For example, the Yahoo News Feed dataset has about 110 billion lines of user-news interaction data; ImageNet contains approximately 14 million labeled images. Due to the limitation of computation resources, single-node ML systems like Theano cannot cope with such large training datasets.

Various distributed ML systems have been proposed based on the Parameter Server (PS) framework [1] to handle big data, such as Petuum [2], MxNet [3], SINGA [4], and TensorFlow [5]. The PS framework can scale to large cluster deployments by having worker nodes performing data-parallel computation, and having server nodes maintaining globally shared parameters. When training ML models, worker nodes continuously pull latest parameters from server nodes, perform computation on partitions of the training dataset, and push generated updates to server nodes. The server nodes would aggregate all received updates, and return a new version of parameters. This is done iteratively to bring parameters closer to the optimal value.

A major challenge of distributed ML is the high communication overhead. Modern ML applications are trending to learn big models with hundreds of millions of parameters. For example, VGG-19 [6] contains 175M parameters, and the multi-class logistic regression (MLR) model trained from Wikipedia consists

of billions of parameters [7]. When using the PS framework, worker nodes push all updates and pulls a huge amount of parameters frequently, generating a large amount of network traffic. Due to the limited bandwidth of commodity cluster network (e.g., 1 Gbps Ethernet), each worker node spends a significant portion of time on communication. Our experiments show that the communication time could be much more than the computation time in many cases. It is important to reduce the communication cost of distributed ML with the PS framework.

Many approaches have been proposed to improve the network performance of distributed ML. For example, MPI-based systems, such as S-Caffe [8], Malt [9] and COTS-HPC [10], use InfiniBand to transmit all parameters or updates with low latency, and leverage CUDA-Aware techniques to improve the performance of GPU-GPU communication in HPC clusters. Bounded Asynchronous Parallel (BAP) models [11] could reduce the frequency of push/pull operations, and overlap communication with computation. More approaches for improving the network performance of distributed ML can be found in Chapter 2.

However, existing approaches have several limitations. First, in many cloud computing platforms and clusters, only commodity network is available, since high-performance network fabrics like InfiniBand could increase infrastructure costs significantly [12]. Second, the primary goal of BAP models is to reduce worker nodes' waiting delay in a PS-based distributed ML system: they do not directly address the network traffic and communication time problem. When the communication time is much longer than the computation time, overlapping communication with computation will not work well.

In this work, we propose Parameter Flow (PF) to tackle the high communication overhead of distributed ML in commodity clusters. PF is a communication layer for the PS framework with two techniques. First, we design an update-centric communication (UCC) model to exchange data between worker nodes and server

nodes using two functions: broadcast and push. Each worker node pushes local updates to the server nodes, and server nodes broadcast aggregated updates to all worker nodes. Compared to the conventional push-pull mechanism, our proposed push-broadcast approach only transmits updates via the network. Second, we develop a dynamic value-bounded filter (DVF). DVF allows worker and server nodes to selectively drop updates during the push and broadcast operations to reduce network traffic and communication time. Meanwhile, DVF guarantees a bound on the magnitude of dropped updates to ensure convergence. As a result, PF can significantly improve the network performance of distributed ML in commodity clusters. We develop a working implementation of PF using ZMQ, and integrate it with several popular ML engines to support a broad range of ML algorithms and models. Extensive experiment results show that PF could speed up popular distributed ML applications by a factor of up to 4.3, compared to a generic PS system using push-pull mechanism.

The rest of this chapter is structured as follows. Section 5.2 presents backgrounds on distributed ML and the PS framework. Section 5.3 shows PF system design. Section 5.4 provides a theoretical analysis of PF. The evaluation results are detailed in Section 5.5. Section 5.6 concludes this chapter.

5.2 Distributed ML on the PS Framework

In this section, we first summarize basic concepts on distributed ML on the PS framework. Table 5.1 shows the notations and definitions used in this chapter.

5.2.1 ML: A Primer

The goal of ML is to learn models from the training dataset, and use them to make predictions on new data. Typically, an ML model consists of a large number of parameters. In this work, we use a vector $\mathbf{W}$ to represent all parameters of an ML

Table 5.1 Summary of notations used in Parameter Flow.

Clock	an integer representing the training progress of a worker node, measured by the number of completed iterations.
$J(\cdot)$	objective function of Problem P1
$f(\cdot)$	loss function of Problem P1
$r(\cdot)$	regularizer function of Problem P1
$\Delta(\cdot)$	function to compute updates
$F(\cdot)$	function to update parameters using received updates
$\mathbf{W}$	vector of parameters of an ML application
$\mathbf{U}$	vector of updates of an ML application
d	size of $\mathbf{W}$ and $\mathbf{U}$
η	initial learning rate
D	training dataset
m	number of examples of the training dataset
p	number of server nodes of a PS-based distributed ML system
n	number of worker nodes of a PS-based distributed ML system
σ	initial bounded value of dropped updates in DVF
δ	initial threshold used to selectively drop updates in DVF

model. An ML application could be described as an optimization problem: given training dataset D with m examples, it tries to find optimum $\mathbf{W}$ to minimize the objective function $J(\cdot)$,

$$\mathbf{P1:} \quad \min_{\mathbf{W}} J(\mathbf{W}) = \sum_{i=1}^m f(\mathbf{W}, D_i) + r(\mathbf{W}), \quad (5.1)$$

where $f(\cdot)$ is the loss function, such as quadratic loss, to represent the prediction error on one example of training dataset, and $r(\cdot)$ is the regularizer function, such as L_1 regularization and L_2 regularization, to limit ML models' complexity. **P1** could be used to represent a wide range of ML models, such as logistic regression (LR), matrix factorization (MF) and convolutional neural networks (CNNs).

It is common to use an iterative-convergent algorithm to solve Problem **P1**, such as gradient descent (GD) and stochastic gradient descent (SGD). An iterative-convergent ML algorithm usually executes following equation iteratively until some convergence criteria are met:

$$\mathbf{U}^t = \Delta(\mathbf{W}^t, D), \text{ then } \mathbf{W}^{t+1} = F(\mathbf{W}^t, \mathbf{U}^t). \quad (5.2)$$

Figure 5.1 System architecture of the Parameter Server framework.

5.2.2 Distributed ML over the PS framework

To cope with big training data sets, many distributed ML systems, such as Petuum, MxNet, Adam, SINGA and TensorFlow, have been proposed. These systems are designed based on the PS framework, and execute data-parallel ML algorithms with pull and push operations. As shown in Figure 5.1, the PS framework has a set of server nodes and a set of worker nodes. Parameters are globally managed on server nodes. Training dataset is partitioned and assigned to worker nodes.

A data-parallel ML algorithm usually executes following equation iteratively on the PS framework with push-pull operations until convergence criteria are met:

$$\begin{aligned}
 \text{Worker Nodes:} \quad & \mathbf{U}_i^t = \Delta(\mathbf{W}^t, D_i), \\
 \text{Server Nodes:} \quad & \mathbf{W}^{t+1} = F(\mathbf{W}^t, \sum_{i=1}^n \mathbf{U}_i^t)
 \end{aligned} \tag{5.3}$$

where i is the index of the i -th worker node. Progress of a worker node is represent by “clock”, which is measured by the number of completed iterations. Updates generated at clock t are timestamped with t . During the training process, a worker node continuously performs computation on the parameter vector $\mathbf{W}$ and outputs an update vector $\mathbf{U}$, which is aggregated on server nodes to update $\mathbf{W}$. To exchange data between worker nodes and server nodes, the PS framework defines a push-pull communication model: worker nodes pull latest $\mathbf{W}$ from server nodes, and push newest $\mathbf{U}$ to server nodes.

Figure 5.2 The BSP and SSP synchronization models. With BSP, there is a synchronization barrier at the end of each iteration. With SSP, worker nodes can use cached parameters to perform computation. In this example, the staleness threshold is 2. Thus, to perform computation of clock 3, a worker node needs to see effects of updates with timestamp 0 from all other worker node.

5.2.3 Synchronization Models

Two types of synchronization models are widely used in the PS framework when executing data-parallel ML algorithms: Bulk Synchronous Parallel (BSP) and Bounded Asynchronous Parallel (BAP).

BSP. BSP places a synchronization barrier at the end of each iteration when executing data-parallel ML algorithms. At clock t , every worker node needs to pull $\mathbf{W}^t$ and push $\mathbf{U}^t$. Server nodes update $\mathbf{W}^t$ to $\mathbf{W}^{t+1}$ after receiving $\mathbf{U}^t$ from all worker nodes. With BSP, distributed ML applications may have high waiting delay, since fast worker nodes must wait for the slow ones at every iteration, as shown in Figure 5.2 (a).

BAP. BAP relaxes the consistency guarantees to reduce the waiting delay, and bounds the amount of inconsistency to guarantee convergence. Since there is no

synchronization barrier, worker nodes can have different clocks. The following two BAP models are widely used in existing distributed ML systems:

- Stale Synchronous Parallel (SSP) [13]. SSP guarantees a bound on the clock difference between the fastest and slowest worker node. It allows a worker node with clock t to use cached parameters, if the cached parameters can see effects of all other nodes' updates from clock 0 to $t - s - 1$, where s is the staleness threshold. In this work, we adopt the latest approach from [13], in which SSP could overlap communication with computation.
- Value-bounded Synchronous Parallel (VAP) [14]. VAP guarantees a bound on the magnitude of the sum of *in-transit* updates from all worker nodes. An update is *in-transit* if it has not been received by every worker node. VAP allows a worker node to perform computation without waiting for other ones, if the magnitude of the sum of all in-transit updates is less than v , where v is a user-defined threshold. However, it is difficult to implement VAP due to its strong condition [15].

Due to the relaxed consistency guarantee, BAP could reduce the communication cost of distributed ML to a certain extent. First, these two models reduce the waiting delay, since fast worker nodes would not always wait for the slow ones. Second, SSP and VAP could reduce the frequency of push/pull operations and overlap communication with computation.

Nevertheless, network communication is still a serious performance bottleneck for PS-based ML applications when learning big models with millions or billions of parameters. The primary goal of BAP models is to reduce the waiting delay of worker nodes. They do not directly reduce the communication time. Thus, when dealing with big ML models, PS-based ML applications can still have prolonged communication time even with BAP models, as shown in Figure 5.2 (b).

5.2.4 Communication Overhead Analysis using AlexNet

To verify the high network communication overhead, we use PS-lite, which is an implementation of PS and a core module of MxNet, to train AlexNet on ImageNet dataset with both BSP and SSP. In the experiment, we use 5 worker nodes and 5 server nodes connected via 1Gbps Ethernet. Each worker node has a NVIDIA K40 GPU. The staleness threshold of SSP is set to 4.

The results show that the communication overhead is quite significant with both BSP and SSP. More specifically, with BSP, each worker node needs to pull and push 61.3 million parameters and updates per iteration. As a result, the five server nodes need to receive and send out around 1226MB data at each iteration. According to our measurement, in an iteration, the communication operation, including pull and push, would take around 6.9s on average. Compared to the computation time, which is around 1.2s per iteration, communication is the dominant component. With SSP, communication consumes around 3.6 times more time than computation through the overall training process. Compared to BSP, the performance gain of SSP comes from the lower waiting delay. However, due to the large network traffic and limited bandwidth, the training job still spends a significant portion of time on communication.

5.3 ParameterFlow System Design

In this section, we describe the system design of ParameterFlow (PF) with two components: UCC and DVF.

5.3.1 System Overview

Figure 5.3 shows the system architecture of PF. Specifically, a PF-enabled PS framework has a similar system architecture to the conventional PS framework.

Figure 5.3 System architecture of a PF-enabled PS framework.

It contains a group of server nodes and a group of worker nodes. Given a distributed ML application, its globally shared parameters are evenly partitioned and assigned to server nodes. The training dataset is partitioned and assigned to worker nodes. During the training process, a worker node performs data-parallel computation its assigned training dataset partition, and pushes generated updates to server nodes, which aggregate them and produce a new version of parameters.

PF is a communication layer for the PS framework, running as a thread on worker and server nodes. To reduce communication cost, PF employs two techniques:

- An update-centric communication (UCC) model. UCC exchanges data between worker and server nodes using two functions: broadcast and push. Specifically, each worker node pushes computed updates to server nodes, and server nodes broadcast aggregated updates to all worker nodes. Worker nodes would update parameters locally based on received aggregated updates. In this way, UCC only transmits updates over the network.
- A dynamic value-bounded filter (DVF). DVF allows worker nodes and server nodes to selectively drop updates during the push and broadcast operations to directly reduce network traffic.

5.3.2 Update-Centric Communication Model

UCC Overview. PF transmits updates with broadcast and push operations. The data-parallel ML algorithm executes following equations iteratively until some convergence criteria are met:

$$\begin{aligned}
 \text{Worker Nodes:} \quad & \mathbf{U}_i^t = \Delta(\mathbf{W}^t, D_i), \\
 \text{Server Nodes:} \quad & \mathbf{U}^{t+1} = \sum_{i=1}^n \mathbf{U}_i^t, \\
 \text{Both Nodes:} \quad & \mathbf{W}^{t+1} = F(\mathbf{W}^t, \mathbf{U}^t).
 \end{aligned} \tag{5.4}$$

At clock t , i -th worker node performs computation on its local set of parameters $\mathbf{W}^t$ and its assigned training dataset partition D_i to generate updates $\mathbf{U}_i^t$, then push $\mathbf{U}_i^t$ to server nodes. Server nodes aggregate received updates to a vector $\mathbf{U}^t$, and update $\mathbf{W}^t$ to $\mathbf{W}^{t+1}$ using function $F(\cdot)$. Instead of letting worker nodes pull $\mathbf{W}^{t+1}$ at clock $t + 1$, PF broadcasts $\mathbf{U}^t$ to all worker nodes, which would update their local $\mathbf{W}^t$ to $\mathbf{W}^{t+1}$ based on the function $F(\cdot)$.

Synchronization Models. UCC can work in conjunction with BSP and BAP. We use SSP as an example of BAP due to its popularity.

UCC-BSP. Server nodes start to broadcast aggregated updates after receiving updates with the same timestamp from all worker nodes. After computing its new version of parameters based on the newly received aggregated updates, a worker node could perform the computation for the next iteration.

UCC-SSP. Worker nodes can use cached stale parameters within a staleness threshold to compute updates. This model could guarantee a bound on the clock difference between the fastest and the slowest worker nodes. It executes the following equation iteratively until some convergence criteria are met:

$$\begin{aligned}
\text{Worker Nodes: } \mathbf{U}_i^t &= \Delta(\mathbf{W}^{\hat{t}}, D_i), \text{ if } t - \hat{t} \leq s \\
\text{Server Nodes: } \mathbf{U}^t &= \sum_{i \in A} \mathbf{U}_i^t + \sum_{i \in B} \sum_{\bar{t}=t+1}^{t+s} \mathbf{U}_i^{\bar{t}}, \\
\text{Both Nodes: } \mathbf{W}^{t+1} &= F(\mathbf{W}^t, \mathbf{U}^t),
\end{aligned} \tag{5.5}$$

where s is the staleness threshold; A is the set of worker nodes which push “guaranteed” updates with timestamp t ; B is the set of worker nodes which push “extra” updates with timestamp range $[t + 1, t + s]$, and $|A| + |B| = n$. Worker nodes in UCC-SSP may be having different clocks, and may generate updates with different timestamps. Server nodes aggregate all received updates regardless of their timestamps. When all worker nodes have pushed updates with timestamp t , the server nodes finalize $\mathbf{U}^t$, and broadcast it. As a result, $\mathbf{W}^{t+1}$ could reflect the changes of two part of updates:

- “guaranteed” updates with a timestamp range of $[0, t]$ from all workers;
- “extra” updates with a timestamp range of $[t + 1, t + s]$ from fast workers.

The worker node would be blocked when its clock minus its parameter version number is larger than the staleness threshold.

UCC Analysis. Compared to the conventional PS framework using push-pull communication mechanism, UCC transmits only updates via the network using push and broadcast operations. Though UCC would not directly improve system performance, it provides opportunities to reduce communication overhead, since updates have more opportunities for reducing communication overhead. For example, when training matrix-parameterized models, all updates are managed in a low-rank matrix $\mathbf{U}$. In this case, $\mathbf{U}$ can be represented by two vectors (i.e. $\mathbf{U} = \mathbf{u}\mathbf{v}^T$), which would significantly reduce the amount of data to be transmitted. If letting worker nodes pulling parameters in each iteration, the low-rank property of $\mathbf{U}$ would be lost during the pull operations [15].

5.3.3 Dynamic Value-Bounded Filter

DVF Overview. DVF allows worker and server nodes to drop updates selectively during the push and broadcast operations, while guaranteeing that the magnitude of dropped updates is less than a bounded value. Given an update vector $\mathbf{U}$, an update is dropped if the corresponding entry in $\mathbf{U}$ is set to 0 by DVF. Thus, PF would not transmit all updates: $\mathbf{U}$ is transformed into a sparse vector, which can be compressed significantly for network transmission. In this way, worker and server nodes can reduce network traffic during the push and broadcast operations.

Motivation. We design DVF to reduce the communication cost of distributed ML based on the following property: data-parallel ML algorithms can converge if we can guarantee a bound on the magnitude of *in-transmit* updates, which are not received by all worker nodes [14]. Thus, worker nodes can perform computation for the next iteration without receiving all updates. Using this property, we design DVF to allow worker and server nodes to push or broadcast updates selectively, and set a bound on the magnitude of dropped updates, which are not transmitted during the push and broadcast operations, to guarantee convergence.

Dropping Updates Selectively. When pushing or broadcasting an update vector $\mathbf{U}^t$, DVF selectively drops some of its entries. Let $\mathbf{U}_{drop}^t$ be the vector containing all dropped entries, which is managed by the local storage of worker or server nodes. Let $\mathbf{U}_{rmn}^t$ be the vector with remaining entries, which would be pushed or broadcast after compression. It should be noted that DVF does not remove $\mathbf{U}_{drop}^t$. In the next communication operation, $\mathbf{U}_{drop}^t$ would be accumulated into the newly generated update vector as follows:

$$\mathbf{U}^{t+1} \leftarrow \mathbf{U}_{drop}^t + \mathbf{U}^{t+1}. \quad (5.6)$$

In this way, DVF would not lose any update information during the training process, so convergence guarantee would not be affected.

Figure 5.4 DVF allows worker nodes and server nodes to selectively drop updates with a given threshold during the push or broadcast operations.

DVF should find $\mathbf{U}_{rmn}^t$ to minimize the amount of remained updates for network transmission, while making sure that the magnitude of $\mathbf{U}_{drop}^t$ is no greater than a bounded value σ^t . It can be formalized as an optimization problem:

$$\mathbf{P2}: \quad \min \quad \|\mathbf{U}_{rmn}^t\|_0, \quad (5.7)$$

$$\text{s.t.} \quad \mathbf{U}^t = \mathbf{U}_{rmn}^t + \mathbf{U}_{drop}^t, \quad (5.8)$$

$$\|\mathbf{U}_{drop}^t\|_2 \leq \sigma^t. \quad (5.9)$$

P2 is an NP-hard problem. To see this, we can transform **P2** into a binary integer programming problem: $\min \|\mathbf{U}^t \times \mathbf{r}\|_0$, subject to $\|\mathbf{U}^t - \mathbf{U}^t \times \mathbf{r}\|_2 \leq \sigma^t$, where $\mathbf{r}$ is a vector of binary variables. If $r_i = 0$, U_i^t is dropped; otherwise, U_i^t is selected for transmission.

We can simplify **P2** by replacing its second constraint with $\sqrt{d}\|\mathbf{U}_{drop}^t\|_\infty \leq \sigma^t$, since $\|\mathbf{U}_{drop}^t\|_2 \leq \sqrt{d}\|\mathbf{U}_{drop}^t\|_\infty$. We can find that the simplified problem can be solved by a $O(1)$ algorithm. Specifically, to minimize the objective function of **P2**, any update whose absolute value is no greater than the threshold $\delta^t = \sigma^t / \sqrt{d}$ should be dropped; and other updated would be kept for transmission.

As shown in Figure 5.4, DVF works on both worker nodes and server nodes to reduce network traffic with δ^t . More specifically, before the push and broadcast operations, DVF selectively drop some updates as follows:

$$\begin{cases} U_{drop,i}^t = U_i^t, \text{ and } U_{rmn,i}^t = 0, \text{ if } |U_i^t| \leq \delta^t \\ U_{rmn,i}^t = U_i^t, \text{ and } U_{drop,i}^t = 0, \text{ if } |U_i^t| > \delta^t \end{cases}, \delta^t = \frac{\sigma^t}{\sqrt{d}}. \quad (5.10)$$

Only U_{rmn}^t would be pushed or broadcast. In this way, DVF would not send all updates during the push or broadcast operations, and could bound the magnitude of dropped updates vector on each node. After compression, we could reduce a lot of network traffic to transmit U_{rmn}^t over network, due to its sparsity.

Time-Varying Threshold. In DVF, the threshold δ^t is a dynamic value based on the training progress of the data-parallel ML algorithms. Specifically, δ^t is time-varying as follows:

$$\sigma^t = \frac{\sigma}{\sqrt{t}}, \text{ and } \delta^t = \frac{\delta}{\sqrt{t}}, \quad (5.11)$$

where σ and δ are the initial bounded value and threshold, t is the clock of a worker node, which works as a regularizer. In particular, t increases δ^t during the early stage of the training, dropping more updates to reduce communication cost. At a later stage, the regularizer decreases δ^t , keeping more tiny updates for network transmission. In Section 5.4, we show that the time-varying threshold δ^t is important to guarantee the convergence of data-parallel ML algorithms.

DVF Analysis. Worker nodes and server nodes need additional computation time to filter, compress and decompress updates during the communication operations. This additional computation overhead would not affect application performance much. Libraries like BLAS/Lapack provide sufficient computation capacity to perform simple filtering operations with multi-thread and SIMD. According to our experiments, snappy can compress and decompress a sparse vector at a rate of up to 950 MB/s per CPU core (2.4 GHz), which would not be the performance bottleneck. Furthermore, each worker node needs additional memory to store all dropped updates. This additional memory usage would not be the performance

bottleneck. For example, the AlexNet model contains 61M parameters. In this case, a worker node needs up to 233MB memory to store all dropped updates. Current commodity server has sufficient memory to store dropped updates.

5.4 Convergence Analysis

In this section, we theoretically analyze PF, and use stochastic gradient descent (SGD) as an example to show how PF affect the convergence of distributed ML. In PF, DVF selectively drops updates for push and broadcast operations. Thus, workers might not be using all latest updates to compute a new set of parameters. This strategy may affect the convergence properties of some distributed ML algorithms. In this section, we focus on analyzing DVF's convergence guarantees. The analysis follows previous work such as [11], [15], [16], [17]. It should be noted that we only consider BSP synchronization model.

We simplify the formulation of an ML problem as follows: $J(\mathbf{W}) = \sum_{i=1}^m f_i(\mathbf{W})$, where $f_i(\mathbf{W}) = f(\mathbf{W}, D^i)$ is a predefined loss function on i -th example of the training dataset. With SGD, we use D^t to compute $\mathbf{W}^t$ at t -th iteration. We use regret to measure the quality of the parameters, which is defined as follows:

$$R[\mathbf{W}] := \sum_{i=1}^T (f_t(\mathbf{W}^t) - f_t(\mathbf{W}^*)), \quad (5.12)$$

where $\mathbf{W}^*$ is the optimum solution. We restrict our discussion to convex and L -Lipschitz loss function $f(\cdot)$. As shown in [11], [15], [16], if $R[\mathbf{W}]/T \rightarrow 0$, our predicted $\mathbf{W}^t$ would converge to the optimum $\mathbf{W}^*$.

To prove the convergence, we use following notations: $\mathbf{U}_i^t$ denotes the computed update vector on i -th worker node; $\mathbf{U}_{i,drop}^t$ is the dropped update vector on i -th worker node; $\mathbf{U}_{i,rnn}^t$ is pushed to server nodes; $\mathbf{U}^t = \sum_{i=1}^n \mathbf{U}_{i,rnn}^t$ denotes the aggregated updates on the server nodes; $\mathbf{U}_{drop}^t$ is the dropped update vector on the server nodes; $\mathbf{U}_{rnn}^t$ is broadcasted to worker nodes. Worker nodes only have

“noisy” parameters (denoted by $\widehat{\mathbf{W}}^t$) since some updates are dropped and would not be received by worker nodes, so that we have $\widehat{\mathbf{W}}^{t+1} = \widehat{\mathbf{W}}^t - \mathbf{U}_{rmn}^t$. Let $\mathbf{W}^t$ denote the vector of “true” parameters produced without dropping any updates. If $t \geq 1$, so that: $\widehat{\mathbf{W}}^t = \mathbf{W}^0 - \sum_{t'=0}^{t-1} \mathbf{U}_{rmn}^{t'}$ and $\mathbf{W}^t = \mathbf{W}^0 - \sum_{t'=0}^{t-1} \sum_{i=0}^n \mathbf{U}_i^{t'}$.

Lemma 5.1. Let $\mathbf{A}^t = \widehat{\mathbf{W}}^t - \mathbf{W}^t$ denote the difference between $\widehat{\mathbf{W}}^t$ and $\mathbf{W}^t$. It follows that:

$$\|\mathbf{A}^t\|_2 \leq (n+1)\delta^t\sqrt{d}, \quad (5.13)$$

Where d is the size of $\mathbf{W}$; n denotes the number of worker nodes; and δ^t is the threshold of DVF at t -th iteration. Lemma 5.1 implies that PF guarantees a bound on the difference between the “noisy” parameters and the “true” parameters. It is an important property to show the convergence of PF.

Proof of Lemma 5.1. Compared to $\widehat{\mathbf{W}}^t$, $\mathbf{W}^t$ contains all dropped updates, thus,

$$\mathbf{A}^t = \widehat{\mathbf{W}}^t - \mathbf{W}^t = \sum_{i=1}^n \mathbf{U}_{i,drop}^{t-1} + \mathbf{U}_{drop}^{t-1}. \quad (5.14)$$

Then, the magnitude of $\mathbf{A}^t$ has an upper-bound,

$$\|\mathbf{A}^t\|_2 \leq \sum_{i=1}^n \|\mathbf{U}_{i,drop}^{t-1}\|_2 + \|\mathbf{U}_{drop}^{t-1}\|_2, \quad (5.15)$$

since $\|\mathbf{a} + \mathbf{b}\|_2 \leq \|\mathbf{a}\|_2 + \|\mathbf{b}\|_2$ for all vectors $\mathbf{a}$ and $\mathbf{b}$. Since DVF guarantees a bound on the magnitude of dropped updates, we get

$$\|\mathbf{U}_{i,drop}^{t-1}\|_2 \leq \sigma^t, i = \{1, 2, \dots, n\}, \text{ and } \|\mathbf{U}_{drop}^{t-1}\|_2 \leq \sigma^t, \quad (5.16)$$

where σ^t is the bound value. Hence,

$$\|\mathbf{A}^t\|_2 \leq (n+1)\sigma^t = (n+1)\delta^t\sqrt{d}. \quad (5.17)$$

This completes the proof of Lemma 5.1.

Theorem 5.1. Given a function $\sum_{i=1}^T f_i(\mathbf{W})$, we use SGD to search for $\mathbf{W}^*$ with learning rate $\eta^t = \eta/\sqrt{t}$. PF would selectively drop some entries of the computed update vector $\mathbf{U}^t$ using DVF with threshold $\delta^t = \delta/\sqrt{t}$, where $\mathbf{U}^t = \eta^t \nabla f_t(\widehat{\mathbf{W}}^t)$. Under suitable conditions ($f(\cdot)$ is convex and L -Lipschitz, the distance between

any two points $D(\mathbf{W}_1 || \mathbf{W}_2) = \frac{\|\mathbf{W}_1 - \mathbf{W}_2\|_2^2}{2} \leq F^2$, we get the regrets as follows:

$$\begin{aligned} R[\mathbf{W}] &:= \sum_{t=1}^T f_t(\widehat{\mathbf{W}}^t) - f_t(\mathbf{W}^*) \\ &\leq \eta L^2 \sqrt{T} + F^2 \sqrt{T} / \eta + 2\delta(n+1)L\sqrt{dT} = O(\sqrt{T}), \end{aligned} \quad (5.18)$$

where F and L are constants.

Theorem 5.1 implies that $\widehat{\mathbf{W}}$ converges to $\mathbf{W}^*$ with PF, since $\frac{R[\mathbf{W}]}{T} \rightarrow 0$. If the initial threshold of DVF δ is equal to 0, PF reduces to native PS-based distributed ML without dropping updates. To guarantee convergence, the value of δ^t should decrease over time. Thus, we make $\delta^t = \delta/\sqrt{t}$ in DVF.

Proof of Theorem 5.1. Since $f_t(\cdot)$ is convex,

$$R[\mathbf{W}] := \sum_{t=1}^T f_t(\widehat{\mathbf{W}}^t) - f_t(\mathbf{W}^*) \leq \sum_{t=1}^T \langle \nabla f_t(\widehat{\mathbf{W}}^t), \widehat{\mathbf{W}}^t - \mathbf{W}^* \rangle. \quad (5.19)$$

Based on Lemma 2 in [16], we can expand $R[\mathbf{W}]$ as follows:

$$R[\mathbf{W}] \leq \eta L^2 \sqrt{T} + \frac{F^2 \sqrt{T}}{\eta} + \sum_{t=1}^T \langle A^t, \nabla f_t(\widehat{\mathbf{W}}^t) \rangle. \quad (5.20)$$

According to L -Lipschitz assumption, i.e. $\|\nabla f_t(\widehat{\mathbf{W}}^t)\|_2 \leq L$, and Lemma 1, we can get the upper-bound of the third term:

$$\sum_{t=1}^T \langle A^t, \nabla f_t(\widehat{\mathbf{W}}^t) \rangle \leq \sum_{t=1}^T (n+1)\delta^t L \sqrt{d} \leq 2\delta(n+1)L\sqrt{dT}. \quad (5.21)$$

Hence, we can get that:

$$R[\mathbf{W}] \leq \eta L^2 \sqrt{T} + \frac{F^2 \sqrt{T}}{\eta} + 2\delta(n+1)L\sqrt{dT} = O(\sqrt{T}). \quad (5.22)$$

This completes the proof of Theorem 5.1.

Figure 5.5 We use a testbed with a tier tree topology to evaluate PF.

5.5 Performance Evaluations

In this section, we evaluate PF’s performance using a working testbed. We first investigate PF’s impact on network traffic. Then, we analyze the communication time and computation time of a PF-enabled distributed ML system. Next, we show PF’s performance improvement.

5.5.1 Experiments’ Configuration

Hardware. We use a testbed with a tier tree topology to conduct experiments. As shown in Figure 5.5, the testbed has 20 servers running 100 Docker containers. Each server contains two Intel Xeon E5-2600 CPUs, 128GB memory and 4TB HDD. The testbed contains 5 NVIDIA Tesla K40 GPU accelerators, which are assigned to 5 containers. Each container has 4 CPU cores, and 20GB memory.

Software. We use Python and C++ to implement a PF-enabled distributed ML system, and make it work in conjunction with three ML engines: MxNet, Theano and Caffe. Specifically, MxNet, Theano and Caffe compute updates on worker nodes. PF exchanges data between worker and server nodes. We also implement two synchronization models on the PF-enabled distributed ML system: BSP and SSP- n , where n is the staleness threshold. In the experiments, we use PS-lite, which is an implementation of the generic PS framework and a core module in

MxNet, as our baseline system. Compared to our PF-enabled distributed ML system, the baseline system has the same computation performance, since it also uses MxNet, Theano and Caffe on worker nodes to perform computation. The key difference is that the baseline system uses push-pull communication model without dropping updates to exchange data between worker and server nodes.

ML Models, Datasets and Configurations. In the experiments, we train three popular ML models, Logistic Regression (LR), Matrix Factorization (MF) and AlexNet, using publicly available datasets. In the following table, we summarize model size, training dataset and the computation engine. To train LR and MF, we use 40 CPU worker nodes and 10 server nodes. To train AlexNet, we use 5 GPU worker nodes and 5 server nodes.

Model	Model Size	Training Dataset	Engine
LR	105M Float32	Criteo Subset (680M ad clicks)	CPU (MxNet)
MF	16.3M Float32	MovieLens (22M movies ratings)	CPU (Theano)
AlexNet	61M Float32	ImageNet Subset (100k images)	GPU (Caffe)

For each model, we use the following values of δ_{up} and δ_{low} :

Threshold	LR	MF	AlexNet
δ_{up}	2×10^{-4}	1×10^{-3}	5×10^{-3}
δ_{low}	1×10^{-4}	5×10^{-4}	2×10^{-3}

These values are derived from history log files. Typically, DVF with δ_{up} roughly drops 60% to 80% of updates on average per push or broadcast operation. The corresponding value for DVF with δ_{low} ranges from 40% to 60%.

5.5.2 Reduced Network Traffic

In this set of experiments, we measure the percentage of reduced network traffic when using PF as the communication layer of the PS framework. To measure the exact network traffic per iteration, we use the BSP synchronization model in the experiments. As shown in Figure 5.6, PF could reduce a large portion of network

Figure 5.6 Percentage of reduced network traffic when using PF to train LR, MF and AlexNet in the testbed with two thresholds.

traffic for the three models, since DVF in PF selectively drops updates during the push and broadcast operations. Compared to the baseline system, PF could reduce up to 80% and 50% of network traffic when training the LR model with δ_{up} and δ_{low} , respectively. For MF, PF can roughly reduce up to 90% and 82% of network traffic with δ_{up} and δ_{low} after the 30-th clock, respectively. The corresponding values for AlexNet with δ_{up} and δ_{low} are 80% and 45%, respectively.

5.5.3 Communication Time vs. Computation Time

In this set of experiments, we measure the communication time and computation time used to train LR, MF and AlexNet to achieve the same convergence criteria. We consider the BSP and SSP synchronization models in the experiments.

As shown in Figure 5.7, PF could significantly reduce the communication time used to train the three ML models when working with both BSP and SSP. For LR, the baseline system with BSP takes 18.3h on communication. In contrast, PF-BSP uses around 4.8h and 8.1h on communication with δ_{up} and δ_{low} . Compared to the baseline system, PF-BSP improves the performance on communication by a factor of up to 3.8 when training LR. The corresponding improvement factors for MF and AlexNet with BSP are 4.1 and 5.4, respectively. PF-BSP with δ_{up} has

Figure 5.7 Communication and computation time used to train LR, MF and AlexNet with various staleness threshold (threshold of 0 denotes BSP).

better communication performance than δ_{low} , since DVF with δ_{up} drops more updates and reduce more communication time correspondingly.

Figure 5.7 shows that PF-SSP also has significant performance improvement on communication. As shown in Figures 5.7 (a-1) (b-1) (c-1), though the baseline

Figure 5.8 PF can speed up distributed ML with BSP and SSP. In the experiments, we use δ_{up} in PF-BSP, and use δ_{low} in PF-SSP-16. The baseline system uses BSP.

system with SSP has better communication performance than the case with BSP, its communication overhead is still significant. For example, when training AlexNet with SSP-16, the baseline system roughly consumes 2.3 times more time on communication than computation. As a comparison, PF-SSP can reduce the communication time to a negligible value with large enough staleness threshold (for example 16). Such significant performance improvement is possible since PF and SSP can work together to overlap communication with computation. As a result, most of push and broadcast operations are performed during computation.

From Figure 5.7, we find that PF reduces the communication time at the cost of higher computation time. For example, when training LR, MF and AlexNet with δ_{up} , PF-BSP could increase computation time by a factor of up to 1.68, 1.42 and 1.75 respectively, compared to the baseline system with BSP. The corresponding values for δ_{low} are 1.22, 1.35 and 1.28. Due to the significant performance gain on communication, the additional computation overhead would not affect the overall training time much.

5.5.4 Application Speedup

Figure 5.8 shows that PF could reduce the overall training time. In this set of

experiments, we use δ_{up} in PF-BSP, and use δ_{low} in PF-SSP-16 to the best of the performance based on the results in Section 5.5.3. We can observe that PF-BSP can reduce the overall training time by a factor of up to 1.6, 1.3 and 1.7 to train LR, MF and AlexNet respectively, compared to the baseline system. When using PF-SSP-16, the corresponding performance improvement factors are 3.1, 1.3 and 3.5, respectively. We should note that PF would not affect the achieved objective values of these three ML applications.

5.6 Summary

In this work, we propose an approach named Parameter Flow (PF) to optimize network performance for distributed ML applications on the PS framework. When training big ML models with the PS framework, worker nodes frequently pull parameters and push updates, resulting in high communication overhead. Our investigations showed that modern distributed ML applications could spend much more time on communication than computation. PF tackles this problem with two techniques: an update-centric communication (UCC) model and a dynamic value-bounded filter (DVF). UCC introduces a broadcast/push model to exchange data between worker nodes and server nodes. DVF reduces network traffic and communication time by selectively dropping updates for network transmission. Extensive performance evaluations showed that PF could reduce communication overhead for distributed ML applications on the PS framework significantly. In the future, we plan to apply PF to more ML applications.

References

- [1] Li, Mu, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. "Scaling Distributed Machine Learning with the Parameter Server." In OSDI, vol. 1, no. 10.4, p. 3. 2014.
- [2] Xing, Eric P., Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak

- Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. "Petuum: A new platform for distributed machine learning on big data." *IEEE Transactions on Big Data* 1, no. 2 (2015): 49-67.
- [3] Chen, Tianqi, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems." *arXiv preprint arXiv:1512.01274* (2015).
- [4] Ooi, Beng Chin, Kian-Lee Tan, Sheng Wang, Wei Wang, Qingchao Cai, Gang Chen, Jinyang Gao et al. "SINGA: A distributed deep learning platform." In *Proceedings of the 23rd ACM international conference on Multimedia*, pp. 685-688. ACM, 2015.
- [5] Abadi, Martín, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems." *arXiv preprint arXiv:1603.04467*(2016).
- [6] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556*(2014).
- [7] Xie, Pengtao, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric P. Xing. "Lighter-Communication Distributed Machine Learning via Sufficient Factor Broadcasting." In *UAI*. 2016.
- [8] Awan, Ammar Ahmad, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K. Panda. "S-Caffe: Co-designing MPI Runtimes and Caffe for Scalable Deep Learning on Modern GPU Clusters." In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 193-205. ACM, 2017.
- [9] Li, Hao, Asim Kadav, Erik Kruus, and Cristian Ungureanu. "Malt: distributed data-parallelism for existing ml applications." In *Proceedings of the Tenth European Conference on Computer Systems*, p. 3. ACM, 2015.

- [10] Coates, Adam, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. "Deep learning with COTS HPC systems." In International Conference on Machine Learning, pp. 1337-1345. 2013.
- [11] Ho, Qirong, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Greg Ganger, and Eric P. Xing. "More effective distributed ml via a stale synchronous parallel parameter server." In Advances in neural information processing systems, pp. 1223-1231. 2013.
- [12] Zhang, Hao, Zhiting Hu, Jinliang Wei, Pengtao Xie, Gunhee Kim, Qirong Ho, and Eric Xing. "Poseidon: A system architecture for efficient GPU-based deep learning on multiple machines." arXiv preprint arXiv:1512.06216 (2015).
- [13] Ho, Qirong, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Greg Ganger, and Eric P. Xing. "More effective distributed ml via a stale synchronous parallel parameter server." In Advances in neural information processing systems, pp. 1223-1231. 2013.
- [14] Wei, Jinliang, Wei Dai, Abhimanu Kumar, Xun Zheng, Qirong Ho, and Eric P. Xing. "Consistent bounded-asynchronous parameter servers for distributed ML." arXiv preprint arXiv:1312.7869 (2013).
- [15] Dai, Wei, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth A. Gibson, and Eric P. Xing. "High-Performance Distributed ML at Scale through Parameter Server Consistency Models." In AAAI, pp. 79-87. 2015.
- [16] Wei, Jinliang, Wei Dai, Abhimanu Kumar, Xun Zheng, Qirong Ho, and Eric P. Xing. "Consistent bounded-asynchronous parameter servers for distributed ML." arXiv preprint arXiv:1312.7869 (2013).
- [17] Dekel, Ofer, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. "Optimal distributed online prediction using mini-batches." Journal of Machine Learning Research 13, no. Jan (2012): 165-202.

Chapter 6

GraphH: Big Graph Analytics in Small Clusters

It is common for real-world applications to analyze big graphs using distributed graph processing systems. Popular in-memory systems require an enormous amount of resources to handle big graphs. While several out-of-core systems have been proposed for processing big graphs using secondary storage, the high disk I/O overhead could significantly reduce performance. In this work, we propose GraphH to enable high-performance big graph analytics in small clusters with limited memory. Specifically, we design a two-stage graph partition scheme to divide the input graph into partitions, and propose a Gather-Apply-Broadcast (GAB) computation model to make each worker process a partition in memory at a time. Moreover, we use an edge cache mechanism to reduce the disk I/O overhead, and design a hybrid strategy to improve the network performance. GraphH can efficiently process big graphs in small clusters or even a single commodity server with limited memory. Extensive evaluations have shown that GraphH could be up to 7.8x faster compared to popular in-memory systems, such as Pregel+ and PowerGraph, and more than 100x faster than recently proposed out-of-core systems, such as GraphD and Chaos.

6.1 Introduction

Many distributed graph processing systems have been proposed to tackle general graph analytics in memory. They follow the “*think as a vertex*” philosophy, and abstract graph computation as vertex-centric programs. More specifically, Pregel [1], Giraph [2], Pregel+ [3], GPS [4], MOCGraph [5] and HuSky [6] adopt the Pregel computation model: they assign the input graph’s vertices to multiple machines, and provide interaction between vertices by message passing along out-edges. PowerGraph [7], PowerLyra [8], GraphX [9] and LFGGraph [10] use the GAS (Gather-Apply-Scatter) model: they split a vertex into multiple replicas, and parallelize the computation for a single vertex in different machines.

Table 6.1 Graph datasets.

Graphs	Vertex Num	Edge Num	Average Degree	Max Indegree	Max Outdegree	Size (CSV)
Twitter-2010	42M	1.5B	35.3	0.7M	770K	25GB
UK-2007	134M	5.5B	41.2	6.3M	22.4K	93GB
UK-2014	788M	47.6B	60.4	8.6M	16.3K	0.9TB
EU-2015	1.1B	91.8	85.7	20M	35.3K	1.7TB

Aforementioned in-memory systems require powerful computation resources to process big graphs (in work, a big graph should contain billions of vertices or hundreds of billions of edges), neglecting the need of average users who are not able to afford a large cluster. During the computation, these systems need to store an entire input graph and all network-transmitted messages in memory. This strategy is appropriate when processing generic graphs with a few billion edges, such as Twitter-2010 and UK-2007 as shown in Table 6.1. Unfortunately, it is common for real-world applications to analyze big graphs such as UK-2014 and EU-2015, which are orders of magnitude larger than Twitter-2010 for example. In this case, the input graph and intermediate messages can easily exceed the memory limit of a cluster, leading to significant performance degradation or even program crashes. We evaluated the memory requirement for running PageRank

Figure 6.1 Memory requirements and execution time for running PageRank on UK-2007 with various distributed graph processing systems. The testbed has 9 servers, and each server contains 12x2.0GHz cores (two Intel Xeon E5-2620 CPUs), 128GB memory, 4x4TB HDDs and 10Gbps Ethernet. We use Giraph-1.1, GraphX-2.0, PowerGraph-2.2 and the newest version of Pregel+, GraphD and Chaos.

on UK-2007 with five in-memory graph processing systems in a 9-node cluster. As shown in Figure 6.1 (a), Giraph, GraphX, PowerGraph, PowerLyra and Pregel+ need 795GB, 685GB, 357GB, 511GB and 281GB memory respectively, indicating 8.5x, 7.3x, 3.8x, 5.5x and 2.9x memory explosions with respect to the input graph's size. To process big graphs like EU-2015, these in-memory systems require a large cluster with at least 5TB memory, approximately.

To solve the memory overflow problem, researchers have recently proposed several out-of-core systems to enable big graph analytics in small clusters with limited memory. Single-node systems, such as GraphChi [11], VENUS [12], X-Stream [13] and GridGraph [14], can process big graphs from secondary storage on a single server. GraphD [15] and Chaos [16] could scale out-of-core graph processing to multiple servers in a cluster. These out-of-core systems typically maintain vertices in memory, and store edges/messages on disks to reduce memory footprint. As shown in Figure 6.1 (a), GraphD and Chaos use 73GB and 26GB memory to run PageRank on UK-2007, respectively. Both systems can process UK-2014 and EU-2015 in a 9-node cluster.

However, existing out-of-core systems could incur a huge amount of costly disk accesses, resulting in significant performance reduction. As shown in Figure 6.1 (a), though each server has 128GB memory, GraphD and Chaos only use 8GB and 3GB memory per server, and cannot efficiently leverage idle memory to reduce disk I/O overhead. For this reason, out-of-core systems usually have much lower performance than most in-memory systems. As shown in Figure 6.1 (b), PowerGraph, PowerLyra and Pregel+ outperform GraphD by 3.3x, 4.8x, and 1.9x, and outperform Chaos by 3.8x, 5.6x and 2.2x, when running PageRank on UK-2007, respectively. Giraph and GraphX, which maintain all data in memory, have lower performance than GraphD and Chaos, since they are implemented based on general-purpose cluster computing frameworks (i.e., Hadoop, Spark), which lack some graph specific optimizations.

In this work, we propose a new distributed graph processing system named GraphH to enable high-performance big graph analytics in a small cluster with limited memory. GraphH is a memory-disk hybrid approach, which does not require storing all data in memory, but maximizes the amount of in-memory data. To achieve this goal, GraphH employs three techniques:

- **Two-Stage Graph Partitioning.** GraphH performs graph partitioning in two stages. In the first stage, GraphH evenly divides the graph into a set of tiles, each of which uses a compact data structure to organize assigned edges. In the second stage, GraphH uniformly assigns tiles to computation servers for running vertex-centric programs.
- **GAB (Gather-Apply-Broadcast) Computation Model.** GraphH uses GAB to represent out-of-core graph computation. During the computation, each vertex maintains a replica on all servers, and each computation worker only loads a tile into memory for processing at a time. GraphH uses following three functions to update a vertex: *Gather* information along in-edges from local memory to compute an accumulator, *Apply* the accumulator to the target vertex, and *Broadcast* new vertex values to other servers over network.

- **Edge Cache Mechanism.** We build an edge cache system to leverage idle memory to reduce disk I/O overhead.

We implement GraphH using C++, MPI, OpenMP, and ZMQ. More specifically, MPI parallelizes vertex-centric computation across multiple servers. OpenMP is used to parallelize the computation across multiple CPU cores in a single server. To improve the communication performance, we use ZMQ to implement a broadcast interface instead of directly using MPI_Bcast. Extensive evaluations have shown that GraphH performs better than existing approaches:

- When processing Twitter-2010 and UK-2007, GraphH outperforms Giraph, GraphX, PowerGraph, PowerLyra and Pregel+ by up to 7.8x.
- When processing UK-2014 and EU-2015, GraphH outperforms GraphD and Chaos by at least 100x.

The rest of this chapter is structured as follows. Section 6.2 describes the system design of GraphH. In Section 6.3, we introduce three techniques to optimize the performance of distributed graph processing. The evaluation results are detailed in Section 6.4. Section 5 concludes this chapter.

6.2 GraphH System Design

In this section, we start with an overview of GraphH system architecture. Then, we present the graph partitioning strategy and data preprocessing engine. Next, we introduce the GAB computation model.

6.2.1 Notations

The input graph $G = (V, E)$ contains $|V|$ vertices and $|E|$ edges. All graphs are directed graphs in this chapter, and it is easy to map an undirected graph to a directed graph. Each vertex $v \in V$ has a unique ID $id(v)$, an incoming adjacency

list $\Gamma_{in}(v)$ and an outgoing adjacency list $\Gamma_{out}(v)$. For simplicity, we use v and $id(v)$ interchangeably. Each vertex v also maintains a value $val(v)$, which may be updated during the computation, and a Boolean field $active(v)$ indicating whether v is active or halted. The in-degree and out-degree of v are denoted by $d_{in}(v)$ and $d_{out}(v)$, where $d_{in}(v) = |\Gamma_{in}(v)|$ and $d_{out}(v) = |\Gamma_{out}(v)|$. If the vertex $u \in \Gamma_{in}(v)$, there is an edge $(u, v) \in E$. In this case, (u, v) is an in-edge of v , and u is an incoming neighbor of v . Similarly, if $u \in \Gamma_{out}(v)$, (v, u) is an out-edge of v , and u is an outgoing neighbor of v . Let $val(u, v)$ denote the edge value of (v, u) . If G is a unweighted graph, $val(u, v) = 1, \forall (u, v) \in E$. The vertex state of v , denoted by $state(v)$, contains different components in various systems. To be consistent, we define $state(v)$ as follows:

$$state(v) = (id(v), val(v), d_{out}(v), d_{in}(v), active(v)).$$

6.2.2 System Architecture

Figure 6.2 shows GraphH system architecture. GraphH contains a distributed file system (DFS), a Spark-based graph pre-processing engine (SPE), and an MPI-based graph processing engine (MPE). Tile is the basic graph processing unit.

DFS. DFS centrally manages all raw graphs (e.g., edge lists), partitioned graphs (i.e., tiles), and results. GraphH can work in conjunction with several widely used DFSs, including HDFS, Ceph and Lustre.

SPE. SPE leverages Spark to split raw graphs into disjoint sets of edges, called tiles, and writes them to DFS.

MPE. MPE takes tiles as input, and runs user-defined computation functions on them in supersteps (or iterations). MPE uses MPI to parallelize the computation across multiple servers, and leverages OpenMP to parallelize the computation across multiple workers in a server. During the computation, each vertex has a

Figure 6.2 GraphH system architecture.

replica on all servers, each worker loads a tile into memory for processing at a time. Thus, MPE can handle big graphs in small clusters with limited memory.

GraphH performs graph partitioning in two stages. In the first stage, SPE evenly splits the input graph's edges into P tiles. In the second stage, MPE uniformly assigns P tiles to N servers before running vertex-centric programs. SPE can be called one time for each input graph, since the pre-processing results (i.e., tiles) are persisted into DFS, and can be reused by MPE to run many specific vertex-centric programs.

6.2.3 Spark-based Graph Pre-Processing

The data pre-processing stage presents following challenges:

- how to evenly distribute the input graph's $|E|$ edges across P tiles;
- how to organize assigned edges in a tile;

Figure 6.3 Graph data pre-processing overview. SPE splits the raw graph into tiles, each of which contains a similar number of edges in CSR format. Edges appear in the same tile as their target vertex.

- how to pre-process big graphs with tens of billions of edges.

To tackle these challenges, we design a Spark-based graph pre-processing engine.

Data Pre-Processing Overview. SPE uses three steps to split the input graph G into P tiles. As shown in Figure 6.3, in the first step, we use a $|V| \times |V|$ sparse matrix to represent G , where the entry in column i and row j is the value of edge (i, j) in G . If G is an unweighted graph, then $val(i, j) = 1$. In the second step, SPE splits the sparse matrix into P tiles in a 1D fashion, each of which roughly holds $S = |E|/P$ nonzero entries. For example, in Figure 6.3, $S = 2, P = 4$. Next, we organize the edges of each tile in the Compressed Sparse Row (CSR) format, and persist them into DFS in binary mode. After these three steps, each tile has following properties:

- Each tile approximately contains $|E|/P$ edges.
- Edges appear in the same tile as their target vertex.
- The target vertices in the same tile have consecutive ids.

Moreover, SPE also computes each vertex's in-degree and out-degree, and store them as two arrays in DFS.

Data Structure. Each tile organizes its edges in an enhanced CSR format. Given a sparse matrix, its basic CSR format consists of three arrays: *val*, *col* and *row*. More specifically, *val* and *col* store all nonzero entries and their column indices

Algorithm 6.1: Spark-based Data Pre-Processing**MapReduce Job 1**

1. `outdegree = edges.map(e  $\Rightarrow$  (e.src, 1)).reduce(SUM)`
2. `indegree = edges.map(e  $\Rightarrow$  (e.src, 1)).reduce(SUM)`

MapReduce Job 2

3. `tile_id = 0, size = 0, splitter = empty`
4. **while** `vertex_id < vertex_number` **do**
5. `size += indegree[vertex_id]`
6. `splitter[tile_id+1] = vertex_id`
7. **if** `size  $\geq S$` **then** *// S is the average tile size*
8. `tile_id += 1, size = 0`
9. **end if**
10. `vertex_id += 1`
11. **end while**

MapReduce Job 3

12. `kv = edges.map(e  $\Rightarrow$  (get tile id(e.target, splitter), e))`
13. `tiles = kv.group_by_key().to_CSR()`
14. `save_to_dfs(outdegree, indegree, tiles)`

in row-major order, respectively. The array *row* records the edge distribution of each vertex: *row*[*i*] and *row*[*i* + 1] indicate the beginning and ending offsets of vertex *i*'s column indices and edge values. If the input graph is unweighted, all its edge values are 1, and its tiles would not manage array *val*.

Tile Size. GraphH allows users to manually configure the average tile size *S* (i.e., number of edges in a tile), where $S = |E|/P$. Since tile is the basic processing unit on MPE, *S* has high impacts on performance. If *S* is too large, GraphH may overflow the memory. For example, if $S = 256M$, each tile consumes more than

1GB memory. Given a physical server with 24 workers (i.e., CPU cores), GraphH must reserve 24GB memory for tiles. Since the power-law distribution of vertex degrees can be observed in most real-world graphs, the size of a tile would be bounded by high-degree vertices, if S is too small. As a result, GraphH cannot evenly split the input graph into tiles, resulting in both storage and computation imbalance. In this work, we configure S to be a value between 15M and 25M to balance the storage and computation requirements. If users could find better tile size for specific applications or datasets, they can update the tile size during the data pre-processing phase.

Splitting Big Graphs on Spark. SPE relies on Spark to pre-process big graphs with three MapReduce jobs, as shown in Algorithm 6.1. The first two MapReduce jobs (line 1-2) return each vertex's in-degree and out-degree. Then, we traverse the in-degree array, assign all encountered vertex's in-edges to a tile until the tile has more than $S = |E|/P$ edges, and store the split information in a *splitter* array. All in-edges of vertex v are assigned to $tile[t_id]$, if $splitter[t_id] \leq v < splitter[t_id + 1]$. We run the third MapReduce job (line 12-14) to group edges by their tile ids, organize them in CSR and flush them to DFS.

SPE Output. SPE converts the input graph into multiple tiles along with an in-degree array and an out-degree array, and use them as the input of vertex-centric programs on MPE. In Table 6.2, we show the input data size of MPE and other graph processing systems. GraphX, PowerGraph and PowerLyra use edge list as input. Pregel+, GraphD, Giraph and Chaos require users to manually convert the input graph to a given format. We observe that SPE also reduces the storage space significantly, other than evenly splitting the input graph. For example, GraphH needs 378GB disk to store EU-2015 in tiles, while its raw edge list needs 1.7TB. The corresponding values of Pregel+, Giraph and Chaos are 862GB, 1.22TB, and 684GB, respectively. Vertex-centric programs can benefit a lot from this compact graph representation with reduced disk I/O overhead when accessing on-disk data.

Table 6.2 Input data size (GB) for different graph processing systems.

Graphs	Edge List (CSV)	Pregel+ GraphD	Giraph	Chaos	GraphH
Twitter-2010	24	12	18	11	7
UK-2007	94	48	69	38	25
UK-2014	874	445	624	351	204
EU-2015	1700	862	1220	684	378

6.2.4 GAB-based Vertex-Centric Computation

We design an MPI-based graph processing engine (MPE) to perform the vertex-centric computation. We start by describing the data layout in MPE. Next, we present the GAB computation model and the parallel computation strategy. The complete pseudo-code description of MPE is shown in Algorithm 6.2.

Data Layout. When running a particular vertex-centric program on MPE, each server maintains three types of data: tiles (partitioned edges), vertex states and messages. MPE uses following strategies to manage these data.

Tiles. Each server stores all assigned tiles on local disk, and uses an edge cache system (detailed in Section 6.3) to store a portion of tiles in memory. MPE uniformly assigns P tiles across N servers as following: i -th tile is assigned to j -th server if $i \pmod N = j$.

Vertices. Each vertex has a replica on all servers. Thus, each server maintains $|V|$ vertices in memory. GraphH uses a list of dense arrays to represent vertex states, and allow users to decide which array to include. For example, each server maintains a rank value array and an out-degree array to run PageRank.

Messages. GraphH manages all messages in memory. As shown in Algorithm 6.2 (line 16), a server only needs to transmit updated vertices to other servers. Therefore, each server uses a $|V|$ -dimensional dense array to manage all messages.

Algorithm 6.2: MPE Computation Engine

```

1. for tile  $\in$  all_tiles do
2.   if tile.id (mode N) = server_id then assigned_tiles.append(tile) end if
3. end for
4. while updated_vertex_num > 0 do
5.   updated_vertex_num = 0
6.   # pragma omp parallel for num_threads(T)
7.   for tile  $\in$  assigned_tiles do
8.     if tile.Bloom_file_contains(updated_vertices) then
9.       load_to_memory(tile)
10.    for v  $\in$  tile.target_vertices do
11.      accum = Gather(v.in_edges, vertex_states)
12.      updated_value = Apply(v.accum, v.value)
13.      if updated_value  $\neq$  v.value then
14.        Broadcast(v, updated_value)
15.    end if, for and while
16. wait_other_servers()
17. for v  $\in$  all_vertices
18.   if v.has_updated_value() then
19.     v.value = v.updated_value
20.     updated_vertex_num += 1
21.   end if
22. end for
23. super_step += 1

```

GAB Model. GraphH abstracts vertex-centric programs into the GAB (Gather, Apply and Broadcast) model. GAB is designed based on the GAS (Gather, Apply and Scatter) model, but differs in both computation and communication. With GAB, each vertex executes three functions to update its value in supersteps: the *gather* function collects information along in-edges and compute an accumulator;

Algorithm 6.3: PageRank Implemented with GAB**Function Gather_PageRank(v.in_edges, vertices)**

1. **for** $e \in v.in_edges$ **do**
2. $accum += e.source.value / vertices.out_degree[u.id]$
3. **end for**
4. **return** $accum$

Function Apply_PageRank(v.in_edges, vertices)

5. $updated_value = 0.15 / num_vertex + 0.85 * v.accum$
6. **return** $updated_value$

Algorithm 6.4: SSSP Implemented with GAB**Function Gather_SSSP(v.in_edges, vertices)**

7. $accum = \infty$
8. **for** $e \in v.in_edges$ **do**
9. $accum = \min(e.source.value + e.value, accum)$
10. **end for**
11. **return** $accum$

Function Apply_SSSP(v.in_edges, vertices)

12. **return** $\min(v.accum, v.value)$

the *apply* function uses the accumulator to produce an updated vertex value; the *broadcast* function copies the updated vertex value to other replicas. The vertex-centric program ends when there are no updated vertices. GAB only requires users to implement the *gather* and *apply* functions. Algorithm 6.3 and 6.4 shows how to use GAB to implement PageRank and single source shortest path (SSSP).

- In PageRank, for each vertex, the *gather* function collects information along its in-edges and sums them to be an accumulator. The *apply* function then produces a new rank value.
- In SSSP, the *gather* function computes the shortest path through each of the in-edges, and the *apply* function returns the new distance.

Parallelized GAB Computation on MPE. MPE performs parallelized out-of-core GAB computation at the level of tiles. As shown in Algorithm 6.2 (line 8 - 17), each server uses T workers (i.e., CPU cores) to process all assigned tiles in each superstep. During the computation, each worker is scheduled to process a tile at a time. When processing a tile, a worker firstly loads it into memory, then traverses it from the first target vertex. For each target vertex, user-defined *gather* and *apply* functions are called to update a vertex. The *gather* function would not incur any network communications, since each vertex has a replica on all servers. Finally, MPE calls the *broadcast* function to copy the updated vertex value to all other replicas. Moreover, MPE follows the Bulk Synchronous Parallel (BSP) synchronization model. As shown in Algorithm 6.2 (line 18), after all servers have completed tile preprocessing, each server updates its vertex values.

Avoid Loading Inactive Tiles. For many algorithms like SSSP, GraphH may just update a few vertices in a superstep. Assume the source vertex list of a tile has no updated vertices, GraphH would not update any target vertices, wasting time to load and process it. To address this problem, GraphH makes each tile leave a Bloom filter in memory to record its source vertex information. When processing a tile, GraphH would first check whether its source vertex list contains any update vertices. If yes, GraphH would continue to load the tile into the main memory for processing. Otherwise, GraphH skips this tile, as shown Algorithm 6.2 (line 10).

GAS vs. GAB. GAB is designed based on the GAS computation model, but differs significantly, as shown in Figure 6.4.

- GAS needs five operations to update a vertex, while GAB only requires a computation operation and a communication operation.
- GAS could reduce vertex replication factor by placing edges intelligently, while GAB requires each vertex to have a replica on all servers. However, GAB does not maintain all edges in memory. Thus it uses much less memory than GAS in practice.

Figure 6.4 GAS vs. GAB. The GAS model requires three computation operations (local sum in the gather phase, update vertex in the apply phase, and active neighbors in the scatter phase) and two communication operations to update a vertex. The GAB model needs one computation operation (gather and apply), and one communication operation (broadcast updated vertex values).

- Given an updated vertex, GAS would activate its neighbors along out-edges. Only active vertices can join the computation of next superstep. In contrast, GAB does not contain this step to save the memory used to store out-edges. At each superstep, GAB makes each vertex checks whether its in-coming neighbors have new values and run the gather, apply functions accordingly.

6.3 System Optimizations

To improve system performance, GraphH uses three techniques: all-in-all vertex replication, edge cache, and hybrid communication.

6.3.1 All-in-All Vertex Replication

GraphH uses an All-in-All (AA) vertex replication policy to manage vertex states across N servers. Specifically, the AA policy requires each vertex has a replica on all servers. Therefore, each server maintains all $|V|$ vertex replicas in memory, even if some of them are used. While the AA policy wastes some memory to store

unused vertices, it can manage each vertex state component in a dense array without indexing overhead. For example, when running PageRank, each server maintains a value array and an out-degree array. Given a vertex v , its position in the array is equal to $id(v)$. Moreover, with the AA policy, GraphH also manages all network transmitted messages in a dense array, which is used to update vertex values at the end of a superstep. Let $M_{i,AA}$ denote the amount of memory required by GraphH for vertex-centric computation in i -th server with the AA policy:

$$M_{i,AA} = Size(Vertex, Msg) \times |V| + Size(Tile) \times T,$$

where $Size(Vertex, Msg)$ is the size of a vertex state and a message. T denotes the number of workers in a server, each of which processes a tile at a time.

The On-Demand (OD) replication policy can avoid storing unused vertex states, but incurs additional indexing overhead. Let $V_{i,DD}$ be the set of vertices managed in i -th server with the OD policy. Specifically, $V_{i,DD}$ only contains the source and target vertices that appear in assigned tiles of i -th server. Each vertex state and its updated value (i.e. message) are indexed and positioned by the vertex id. With the OD policy, the amount of required memory in i -th server is:

$$M_{i,OD} = Size(ID, Vertex, Msg) \times |V_{i,DD}| + Size(Tile) \times T.$$

Assume the input graph G is a random graph: the neighbors of a vertex in G are randomly chosen. In this case, $|V|$ vertices and $|E|$ edges are evenly assigned to N servers. The expected number of vertices maintained by i -th server is:

$$E[|V_{i,od}|] \leq \left(1 - \left(1 - \frac{d_{avg}}{|V|} \right)^{\frac{|V|}{N}} \right) |V| + \frac{|V|}{N},$$

where $(1 - (1 - d_{avg}/|V|)^{|V|/N})|V|$ is the number of source vertices, $|V|/N$ is the amount of target vertex, and some target vertices may appear in the source vertex list. For big graphs, due to $\lim_{n \rightarrow \infty} \left(1 - \frac{1}{n} \right)^n = e^{-1}$, we have

$$E[|V_{i,od}|] \leq \left(1 - e^{-\frac{d_{avg}}{N}} \right) |V| + \frac{|V|}{N}.$$

Figure 6.5 GraphH memory usage analytics. (a) shows the expected memory usage per server, when running PageRank with the All-in-All policy and On-Demand policy. (b) shows the memory usage of GraphH (All-in-All policy without edge cache) per server in a 9-node cluster to run PageRank and SSSP. Each server has 128GB memory.

We take PageRank as an example to show that the AA policy is more memory efficient than the OD policy in small clusters, since it eliminates the indexing overhead at the cost of storing some unused vertices and messages. Specifically, with the AA policy, $Size(Vertex, Msg) = 20 \text{ bytes}$, since each vertex value and message is represented by a double-precision number (8 bytes), and each vertex out-degree is represented by an integer (4 bytes). When using the OD policy, $Size(ID, Vertex, Msg) = 24 \text{ bytes}$, since each vertex id is represented by an unsigned integer (4 bytes). Assume that Twitter-2010, UK-2007, UK-2014 and EU-2015 are random graphs, Figure 6.5 (a) shows the expected memory usage per server. We can observe that the AA policy consumes less memory than the OD policy for all graphs in a small cluster with less than 16 servers. In a big cluster with more than 48 servers, the OD policy consumes less memory than the AA policy to run PageRank on EU-2015. Since GraphH is designed for big graph analytics in small clusters, it uses the AA policy for memory-efficiency.

6.3.2 Edge Cache

GraphH uses an edge cache system to reduce disk I/O overhead. As shown in Figure 6.5 (b), when running PageRank on EU-2015 in a 9-node cluster, each

Table 6.3 Compression ratio and processing throughput per CPU core.

Graphs	Compression Ratio			Throughput (MB/s)		
	snappy	zlib-1	zlib-3	snappy	zlib-1	zlib-3
Twitter-2010	1.75	2.78	3.22	870	55	46
UK-2007	1.89	3.71	4.54	947	58	53
UK-2014	1.96	4.34	5.26	903	65	50
EU-2015	1.96	4.35	5.88	890	62	56

server only uses 33GB memory, leaving other 95GB memory idle. To reduce the amount of costly disk accesses, we build a tile cache system on these idle memory resources. During the vertex-centric computation, when a worker needs to load a tile, it firstly searches the cache system. If hit, the worker can get the target tile without disk I/O operations. Otherwise, the worker reads the target tile from local disks, and leaves it in the cache system if the cache system is not full.

To further reduce the disk I/O overhead, GraphH can compress tiles in the edge cache system. Table 6.3 shows that popular compressors, such as snappy and zlib, can efficiently reduce the data size of real-world graphs. For example, zlib-3 (N denotes the compression level of zlib in $\text{zlib-}N$) can compress EU-2015 tiles by a factor of 5.88, and reduce its data size to 62GB. While workers need additional decompression time, GraphH's edge cache system still provides much higher performance than hard disks. Table 6.3 shows that snappy can decompress tiles at a rate of up to 903MB/s per CPU core. The corresponding value of zlib-3 is 56MB/s. If a server has 22 workers, its tile loading rate of zlib-3 is about 1.2GB/s. As a comparison, we can only achieve up to 310MB/s sequential disk read speed with RAID5 in our testbed, and the disk bandwidth is shared by all workers (i.e., CPU cores) of a server.

GraphH's edge cache system can automatically switch to the most suitable mode, considering disk I/O and decompression overhead. In this work, we consider following four cache modes:

Figure 6.6 Execution time and cache hit ratio comparison with different cache modes. We run PageRank on EU-2015 in this set of experiments.

- Mode-1: Cache uncompressed tiles.
- Mode-2: Cache compressed tiles processed by snappy.
- Mode-3: Cache compressed tiles processed by zlib-1.
- Mode-4: Cache compressed tiles processed by zlib-3.

When having limited memory, it is important to select compressors with high compression rate to reduce disk I/O overhead. Figure 6.6 shows that mode-3 could improve the system performance by a factor of 17.6 by caching all tiles in memory than mode-1 when running PageRank on EU-2015 with 3 servers. When having the same cache hit ratio, the decompression overhead can reduce the system performance. Figure 6.6 shows that mode-4 increases the execution time by a factor of 2 when using 9 servers, compared to mode-1.

To balance the disk I/O overhead and the decompression overhead, GraphH could select the most suitable cache mode at the beginning a program. Let C denote the cluster's edge cache capability, S is the input graph's tile size, and let γ_i be the estimated compression ratio is using cache mode- i . GraphH tries to minimize i constrained by $S/\gamma_i \leq C$. If no cache mode can satisfy this constraint, GraphH uses mode-3. In this work, $\gamma_0 = 1$, $\gamma_1 = 2$, $\gamma_2 = 4$, $\gamma_3 = 5$, according to our collected data shown in Table 6.4.

Table 6.4 Graph data size in tiles after compression.

Graphs	Edge List (CSV)	Tile (Raw)	Tile (snappy)	Tile (zlib-1)	Tile (zlib-3)
Twitter-2010	24GB	6.5GB	3.7GB	2.3GB	2GB
UK-2007	94GB	23GB	12GB	6.2GB	5GB
UK-2014	874GB	196GB	100GB	45GB	37GB
EU-2015	1700GB	362GB	185GB	80GB	62GB

6.3.3 Hybrid Communication

We find that a single communication mode cannot perform well for vertex-centric programs. Specifically, during the computation, each worker generates a set of updated vertex values when processing a tile. GraphH makes each worker buffer updated values, and broadcast them to other servers in a single message after the completion of the whole tile processing. It is advantageous to use a dense array representation for updated vertex values along with a bitvector to record updated vertices. However, this dense communication mode may waste a lot of network bandwidth when a few of vertices are updated, because it needs to send many zero values. For example, Figure 6.7 (a) shows that less than 50% vertices are updated after the 160th superstep when running PageRank on UK-2007 in a 9-node cluster. The sparse array representation, which converts a dense array into a list of indices and values, can solve this problem, since it only sends updated vertices. However, sparse communication mode would waste a huge amount of network resources to send indices, if the vertex updated ratio is high. As shown in Figure 6.7 (b), only after the 160th superstep, the sparse mode can have lower network traffic than the dense mode.

We design a hybrid communication mode to reduce the network traffic and improve the communication performance of GraphH. Specifically, GraphH firstly uses a dense array to store all updated vertex values. Before the broadcasting phase, GraphH checks its sparsity ratio. If the sparsity ratio is higher than a given

Figure 6.7 Network traffic of PageRank on UK-2007 in a 9-node cluster. In (a), we show the vertex update ratio (i.e., the percentage of vertices which update their values) of a superstep during the computation. In (b), we compare the network traffic size of a superstep between sparse and dense communication mode. In (c), we show the network traffic size of a superstep when using the hybrid communication mode. In (d), we show the execution time of a superstep with hybrid communication mode.

threshold (in this work, this threshold is set to 0.8), GraphH converts it into a sparse array, which only stores non-zero values and their indices. For example, in Figure 6.7 (c), at the beginning of the program, GraphH would broadcast messages under the dense node. At the end of the program, GraphH would switch to the sparse communication mode to avoid sending zeros.

Message compressing can further improve communication performance. Figure 6.7 (c) shows that snappy, zlib-1 and zlib-3 can reduce network traffic by a factor of 1.7, 2.3 and 2.3, respectively. Figure 6.7 (d) shows that reduced network traffic can improve the graph processing performance. In the first 50 supersteps, GraphH

roughly takes 2.32s per superstep without compression. When using snappy, zlib-1 and zlib-3, GraphH averagely takes 1.73s, 1.56s and 1.5s per superstep. While zlib could save more network traffic, the additional decompressing overhead makes it consume more execution time than snappy. The default network traffic compressor of GraphH is snappy.

6.4 Performance Evaluations

In this section, we evaluate GraphH’s performance using a testbed with two graph applications: PageRank, SSSP. The hardware and software configurations are same with the testbed shown in Figure 6.1. We use the average execution time per superstep as the performance metric. For each experiment, we run 11 supersteps, and calculate the average execution time without the first superstep, since distributed graph processing systems usually load the input graph to memory during this superstep. In the experiments, we do not compare the graph loading time, since existing graph processing systems do not have unique storage system requirements. For example, Pregel+ and GraphD only support HDFS. In Chaos, we must manually split the input graph into partitions, and distributed them to multiple servers. The performance of each distributed graph processing system could be printed using Glog, the C++ implementation of the Google logging module. Specifically, we use Glog to print execution time and collected network data into a file every superstep.

In the experiments, we use four real-world graphs as inputs: Twitter-2010, UK-2007, UK-2014 and EU-2015. Following table shows their vertex/edge number and the data size in CSV format.

	Twitter-2010	UK-2007	UK-2014	UK-2015
Vertex Num	42M	134M	788M	1.1B
Edge Num	1.5B	5.5B	47.6B	91.8B
Data Size (CSV)	24GB	94GB	874GB	1.7TB

Figure 6.8 In-degree and out-degree distribution of Twitter-2010, UK-2007, UK-2014 and EU-2015. These four graphs follow skewed power-law distribution.

In Figure 6.8, we plot the in-degree and out-degree distribution of these four real-world graphs. We can observe that these four graphs follow skewed power-law degree distribution: most vertices have relatively few neighbors while a few have many neighbors (e.g., celebrities in a social network).

6.4.1 Effect of Compressed Edge Caching

We analyze the effect of compressed edge caching of GraphH in this experiment. Specifically, we run PageRank and SSSP over largest graph in this chapter, EU-2015, on a cluster with 9 servers, and measure the execution time and memory footprint of GraphH with different edge caching modes.

Table 6.5 Average execution time per superstep to run PageRank and SSSP on EU-2015 using 9 servers with different edge cache modes. We collect memory usage of each server to show relationship between execution and memory footprint.

Edge-Cache Mode	PageRank		SSSP	
	Execution Time (s)	Memory Usage (GB)	Execution Time (s)	Memory Usage (GB)
No Cache	411.2 $\pm$ 65.7	18.2	309.6 $\pm$ 157.3	11.7
Mode-1	9.8 $\pm$ 0.8	62.5	2.1 $\pm$ 0.8	55.3
Mode-2	11.2 $\pm$ 1.1	43.7	3.8 $\pm$ 1.3	36.1
Mode-3	17.1 $\pm$ 1.5	28.8	11.6 $\pm$ 3.1	20.9
Mode-4	19.8 $\pm$ 2.6	25.5	13.7 $\pm$ 3.7	17.6

Table 5 shows the average execution time (per superstep) of running PageRank and SSSP on EU-2015 with different edge cache modes, and memory usage of each server. We can observe that GraphH’s compressed edge caching mechanism could significantly improve graph processing performance. Specifically, if we do not use cache, GraphH should fetch all required edges from disk at each superstep, resulting in high disk I/O overhead. In this case, GraphH needs 411.2s and 309.6s to complete a superstep of PageRank and SSSP, respectively. If we set up Mode-1 edge cache system, all edges could be stored in memory for both applications. Each server roughly needs 62.5GB and 55.3GB memory for PageRank and SSSP. In this case, GraphH only needs 9.8s and 2.1s on average to complete a superstep of PageRank and SSSP. If we do not have enough memory, GraphH’s edge cache mechanism could compress cached edges. For example, when using Mode-3, GraphH needs 17.1s and 11.6s on average to complete a superstep. In this case, each server only consumes 28.8GB and 20.9GB memory.

6.4.2 PageRank Performance

As shown in Figure 6.9, GraphH could achieve much higher performance than Pregel+, PowerGraph, PowerLyra, GraphD and Chaos when running PageRank on all four input graphs. Moreover, we observe that the memory management

Figure 6.9 Average execution time per superstep to run PageRank on four graph datasets (Twitter-2010, UK-2007, UK-2014 and EU-2015) using six distributed graph processing systems (GraphH, Pregel+, PowerGraph, PowerLyra, GraphD and Chaos). In (c) (d), we only show the performance of GraphH, GraphD and Chaos, since Pregel+, PowerGraph, PowerLyra cannot run PageRank on UK-2014 and EU-2015.

strategy of GraphH is efficient, since it can even process big graphs like UK-2014 and EU-2015 in a single node.

As shown in Figure 6.9 (a) (b), when running PageRank on Twitter-2010 using 9 servers, GraphH outperforms Pregel+, PowerGraph, PowerLyra, GraphD and Chaos by 7.8x, 6.3x, 5.3x, 13x and 25x, respectively. The corresponding speedup ratios for UK-2007 are 7.5, 4.3, 3.5, 18 and 19. For Twitter-2010 and UK-2007, Pregel+, PowerGraph and PowerLyra can maintain all vertices, edges and messages in memory. GraphH can demonstrate higher processing performance than these in-memory systems due to the reduced communication overhead.

Figure 6.10 Average execution time per superstep to run SSSP on four graph datasets (Twitter-2010, UK-2007, UK-2014 and EU-2015) using six distributed graph processing systems (GraphH, Pregel+, PowerGraph, PowerLyra, GraphD and Chaos). In (c) (d), we only show the performance of GraphH, GraphD and Chaos, since Pregel+, PowerGraph, PowerLyra cannot run SSSP on UK-2014 and EU-2015.

As shown in Figure 6.9 (c) (d), GraphH could efficiently run PageRank on big graphs like UK-2014 and EU-2015 in a small cluster or even a single server. More specifically, in a single node, GraphH only takes 68s and 131s to run a superstep of PageRank on UK-2014 and EU-2015. When having a cluster with 9 servers, GraphH only needs 7.5s and 10s on average to execute one superstep, and could outperform GraphD and Chaps by 320 and 110, respectively.

6.4.3 SSSP Performance

Figure 6.10 shows that GraphH works well with SSSP. When running SSSP on

Twitter-2010 and UK-2007 with 9 servers, GraphH has a similar performance with Pregel+, and both of them roughly takes 0.4s to run a superstep. The reason is that the communication overhead is not significant for SSSP, since only a small partition of vertices may update their values. GraphH outperforms PowerGraph and PowerLyra by up to 2 for running SSSP on Twitter-2010 and UK-2007.

Figure 6.10 (c) (d) shows that GraphH's memory management is efficient when running SSSP on big graphs. Specifically, GraphH roughly takes 1.3s and 2.1s to execute a superstep of SSSP on UK-2014 and EU-2015 with 9 servers. Based on collected profile data, the performance comes from three techniques: 1) GraphH could load all edges in memory, so that it would incur costly disk I/O operations during the computation; 2) GraphH could avoid processing inactive tiles using Bloom filters, so that it could reduce unnecessary computation time; 3) GraphH uses a hybrid communication mode to reduce network traffic and communication time. While GraphD and Chaos could run SSSP on UK-2014 and EU-2015, due to their high disk I/O overhead, GraphH could outperform them by at least 350x.

6.5 Summary

In this work, we tackle the challenge of big graph analytics in small clusters with limited memory. Existing in-memory systems need a huge amount of resources to handle big graphs, and out-of-core systems have poor performance due to high disk I/O overhead. We propose a new distributed graph processing system named GraphH, which does not require to store all data in memory, but it maximizes the amount of in-memory data. GraphH partitions the input graph into tiles, and makes each worker process a tile in memory at a time to reduce memory footprint. We design an edge cache system to reduce disk I/O overhead, and use a hybrid approach to reduce communication overhead. Extensive evaluations show that GraphH outperforms existing in-memory systems by up to 7.8x, and outperforms existing out-of-core systems by more than 100x.

References

- [1] Malewicz, Grzegorz, Matthew H. Austern, Aart JC Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: a system for large-scale graph processing." SIGMOD 2010, pp. 135-146. ACM, 2010.
- [2] Ching, Avery, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. "One trillion edges: Graph processing at facebook-scale." Proceedings of the VLDB Endowment 8, no. 12 (2015): 1804-1815.
- [3] Yan, Da, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. "Pregel algorithms for graph connectivity problems with performance guarantees." Proceedings of the VLDB Endowment 7, no. 14 (2014): 1821-1832.
- [4] Salihoglu, Semih, and Jennifer Widom. "Gps: A graph processing system." In Proceedings of the 25th International Conference on Scientific and Statistical Database Management, p. 22. ACM, 2013.
- [5] Zhou, Chang, Jun Gao, Binbin Sun, and Jeffrey Xu Yu. "Mocgraph: scalable distributed graph processing using message online computing." Proceedings of the VLDB Endowment 8, no. 4 (2014): 377-388.
- [6] Yang, Fan, Jinfeng Li, and James Cheng. "Husky: Towards a more efficient and expressive distributed computing framework." Proceedings of the VLDB Endowment 9, no. 5 (2016): 420-431.
- [7] Gonzalez, Joseph E., Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs." In OSDI, vol. 12, no. 1, p. 2. 2012.
- [8] Chen, Rong, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. "Powerlyra: Differentiated graph computation and partitioning on skewed graphs." In Proceedings of the Tenth European Conference on Computer Systems, p. 1. ACM, 2015.
- [9] Gonzalez, Joseph E., Reynold S. Xin, Ankur Dave, Daniel Crankshaw,

- Michael J. Franklin, and Ion Stoica. "GraphX: Graph Processing in a Distributed Dataflow Framework." In OSDI, vol. 14, pp. 599-613. 2014.
- [10] Hoque, Imranul, and Indranil Gupta. "Lfgraph: Simple and fast distributed graph analytics." In Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems, p. 9. ACM, 2013.
- [11] Kyrola, Aapo, Guy E. Blelloch, and Carlos Guestrin. "Graphchi: Large-scale graph computation on just a pc." USENIX, 2012.
- [12] Cheng, Jiefeng, Qin Liu, Zhenguo Li, Wei Fan, John CS Lui, and Cheng He. "VENUS: Vertex-centric streamlined graph computation on a single PC." In Data Engineering (ICDE), 2015 IEEE 31st International Conference on, pp. 1131-1142. IEEE, 2015.
- [13] Roy, Amitabha, Ivo Mihailovic, and Willy Zwaenepoel. "X-stream: Edge-centric graph processing using streaming partitions." In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 472-488. ACM, 2013.
- [14] Zhu, Xiaowei, Wentao Han, and Wenguang Chen. "GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning." In USENIX Annual Technical Conference, pp. 375-386. 2015.
- [15] Yan, Da, Yuzhen Huang, James Cheng, and Huanhuan Wu. "Efficient processing of very large graphs in a small cluster." arXiv preprint arXiv:1601.05590 (2016).
- [16] Roy, Amitabha, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. "Chaos: Scale-out graph processing from secondary storage." In Proceedings of the 25th Symposium on Operating Systems Principles, pp. 410-424. ACM, 2015.

Chapter 7

Summary and Future Work

In this chapter, we summarize this thesis and discuss the future work for performance improvement of distributed machine learning and graph processing at scale over virtualized infrastructure.

7.1 Summary

In this thesis, we focus on the performance optimization for distributed machine learning and graph processing (MLGP) at scale over virtualized infrastructure. The major contributions of this thesis are summarized as follows:

- To improve cluster utilization, we design an efficient cluster management system called Dorm using container-based virtualization techniques. Dorm could improve cluster utilization by a factor of up to 2.32 and thus accelerate distributed MLGP applications by a factor of up to 2.76, compared to existing cluster management systems with static resource allocation models.
- To improve the metadata lookup performance for large-scale distributed file systems, we design a scalable metadata lookup service called MetaFlow using Software-defined networking (SDN) techniques. MetaFlow could reduce metadata-related operation latency by a factor of up to 5, and thus speed up distributed MLGP applications by a factor of up to 1.81, compared to existing DHT-based metadata lookup services.
- To reduce the communication overhead of large-scale distributed machine learning, we design a novel communication layer for the Parameter Server framework called ParameterFlow, which could speed up popular distributed ML applications by a factor of up to 4.3, compared to existing PS framework.
- To enable big graph analytics in small clusters with limited memory, we design a new distributed graph processing system called GraphH, which could outperform existing distributed in-memory systems by up to 7.8x, and outperform existing distributed out-of-core systems by more than 100x.

7.1.1 Platform Optimizations

In Chapter 3, we propose a cluster management system named Dorm to share a cluster with multiple and diverse distributed MLGP workloads efficiently and fairly. Currently, organizations are trending to run multiple distributed MLGP

systems (e.g., MxNet, TensorFlow, Pregel, PowerGraph) in the same cluster, so that users can pick the most suitable one for each specific application. To achieve this goal, many cluster management systems have been proposed, such as Mesos and Yarn. However, existing approaches can only allocate a static partition of the cluster to each distributed MLGP application, leading to poor resource utilization. To tackle this problem, our proposed cluster management system Dorm employs two techniques: a dynamically-partitioned cluster management mechanism and a utilization-fairness optimizer. With Dorm, each distributed MLGP application would run in a dynamic partition of the cluster. When detecting newly submitted or completed applications, Dorm's utilization-fairness optimizer would make new resource allocation decisions, and reallocates existing resources allocations accordingly to consistently keep high utilization and low fairness loss. We have implemented Dorm and enabled it to support Petuum, MxNet, TensorFlow, Caffe, GraphH and GraphD. Extensive performance evaluations have shown that Dorm could simultaneously increase the resource utilization by a factor of up to 2.32, reduce the fairness loss by a factor of up to 1.52, and speed up popular distributed MLGP applications by a factor of up to 2.76, compared to existing approaches.

In Chapter 4, we propose a distributed metadata lookup service named MetaFlow for DFSs. Existing large-scale DFSs usually use DHT to manage their metadata. Popular DHT-based systems contain a lookup subsystem and a storage subsystem. When performing a metadata operation like open and read, users should first use the lookup subsystem to locate the desired metadata object, then fetch it from the corresponding metadata server. These two subsystems may compete for CPU resources, which leads to reduced system throughput and high latency. MetaFlow solves this problem by using the SDN techniques to transfer the metadata lookup service to the network layer. MetaFlow implements this approach by mapping a data center's network topology to a logical B-tree. Appropriate flow tables will then be generated and distributed to SDN-enabled switches. Thus, the switches would have the ability to forward metadata requests to the corresponding server

directly. Experiments show that MetaFlow could increase the system throughput by a factor of up to 6.5, and reduce the system latency by a factor of up to 5 for the metadata management, compared to DHT-based approaches.

7.1.2 Framework Optimizations

In Chapter 5, we propose an approach named Parameter Flow (PF) to reduce the communication overhead of large-scale distributed machine learning (ML). The Parameter Server (PS) framework is widely used to train ML models in parallel. It can handle big training datasets by having a group of worker nodes performing data-parallel computation, and having a set of server nodes maintaining globally shared parameters. When training ML models, worker nodes need to frequently pull parameters from server nodes and push updates to server nodes, resulting in high communication overhead. Modern distributed ML applications could spend much more time on communication than computation. To address this problem, our proposed PF employs two main techniques: an update-centric communication (UCC) model and a dynamic value-bounded filter (DVF). Specifically, UCC introduces a broadcast/push model for worker and server nodes to exchange data, and it only transmits updates over network. DVF could directly reduce network traffic and communication overhead by selectively dropping updates for network transmission. Meanwhile, we show that DVF could guarantee the convergence of distributed ML applications. Extensive performance evaluations showed that PF could speed up popular distributed ML applications by a factor of up to 4.3, compared to the conventional PS framework.

In Chapter 6, we design GraphH to solve the memory overflow problem for large-scale distributed graph processing. Specifically, existing in-memory distributed graph processing systems (e.g., Pregel, Giraph, PowerGraph) need a huge amount of resources to store all input data and intermediate results in memory during the computation. When processing big graphs with billions of vertices or hundreds

of billions of edges, these approaches can easily exceed the memory capacity of a typical cluster. While existing out-of-core systems (e.g., Chaos, GraphD) could process big graphs from the secondary storage, they have poor performance due to the high disk I/O overhead. To enable high-performance big graph analytics in small clusters with limited memory, our proposed distributed graph processing system GraphH does not require to store all data in memory, but it maximizes the amount of in-memory data. Specifically, GraphH partitions the input graph into tiles, and makes each worker process a tile in memory at a time to reduce memory footprint. We design an edge cache system to reduce disk I/O overhead, and use a hybrid approach to reduce communication overhead. Extensive evaluations show that GraphH outperforms existing in-memory systems by up to 7.8x, and outperforms existing out-of-core systems by more than 100x.

7.2 Future Work

In the future, we will extend the work in the following aspects.

7.2.1 Resource-Efficient and QoS-Aware Cluster Management

In this thesis, we only considered cluster utilization and fairness when allocating cluster resources to distributed MLGP applications. In practice, it is also crucial for a cluster management system to meet the Quality of Service (QoS) constraints for each workload. Specifically, QoS is a measure of the overall performance experienced by a specific user of a computer system. In this work, it refers to the processing time of a distributed MLGP application. In a QoS-aware cluster, users need to specify the performance constraints for each workload, and would not specify the maximal and minimal amount of resources for each application. The cluster management system determines the right amount of resources allocated to each workload to meet these constraints. In the future, we plan to leverage deep learning techniques to predict the resource needs for each workload, and let the

cluster management system adjust existing resource allocations and assignments to maximize cluster utilization and meet performance constraints.

7.2.2 Efficient Large-Scale Graph Processing on a Single Machine

Recent studies have shown that single-machine graph processing systems can be as highly competitive as cluster-based systems on large-scale problems in many cases. Specifically, out-of-core systems, such as GraphChi, X-Stream, VENUS and GridGraph, provide cost-effective solutions for big graph analytics on a single machine. These systems usually break the input graph into a set of small shards, and process a shard in memory at a time during the computation. There will be a huge amount of disk accesses, which can be the performance bottleneck. To reduce the disk I/O overhead, many computation models have been proposed, such as the parallel sliding window model of GraphChi, the edge-centric scatter-gather model of X-Stream, the vertex-centric streamlined processing model of VENUS, and the dual sliding windows model of GridGraph. However, current out-of-core systems still have low performance due to the high disk I/O overhead. In the future, we plan to design an efficient single-machine big graph analytics system based on our existing work GraphH. First, we plan to propose a new out-of-core graph computation model to avoid reading and writing vertices on disk. After that, we would use a more efficient edge cache mechanism to fully utilize the available memory of a single machine to further reduce the amount of disk accesses. We would also consider leveraging heterogeneous storage devices (e.g., SSD) and computation devices (e.g., GPU) to improve the system performance.

7.2.3 High-Performance Time-Evolving Graph Analytics at Scale

Time-evolving graphs are arising in many application domains, such as social networks and Internet-of-Things (IoT). In the future, we plan to design a time-evolving graph processing system based on the lambda architecture, which is

widely used in general-purpose data-processing systems. To balance throughput, latency, and accuracy, we plan to use two engines to process real-time and batch graph data. First, we use an incremental processing engine to process arrived data streams in one pass, and update affected vertex values immediately at the cost of losing some accuracy. Second, we use an iterative processing engine to obtain accurate vertex values. After that, we could process real-time big graphs with low latency and high throughput.