Copyright
by
Gurbinder Singh Gill
2020

The Dissertation Committee for Gurbinder Singh Gill
certifies that this is the approved version of the following dissertation:

Compiler and System for Resilient Distributed Heterogeneous Graph Analytics

Committee:

Keshav Pingali, Supervisor

Donald S. Fussell

Christopher J. Rossbach

Ramesh Peri

Todd Mytkowicz

Compiler and System for Resilient Distributed Heterogeneous Graph Analytics

by

Gurbinder Singh Gill

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2020

Dedicated to my parents Devinder and Bakhshish Gill, my sister Parminder Gill and my wife Anvita Seth for their unconditional love, endless support and encouragement which made this incredible journey possible.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my research supervisor Prof. Keshav Pingali for giving me the opportunity to do research, providing continuous support, resources and invaluable guidance throughout this research. His hard work, clear vision, sincerity and motivation has deeply inspired me. I feel extremely indebted to him for his relentless efforts to help us improve our all-round skills as researchers and bring the best in us. I admire his patience to sit with us till 4 am for our anywhere on earth paper submission deadlines and endless practice rounds of presentations. I could not have imagined having a better advisor and mentor for my Ph.D. Thank you, Prof. Keshav Pingali, for the most amazing Ph.D. journey ever.

I would like to express my heartfelt appreciation to my labmates in my research group, the Intelligent Systems and Software group at UT Austin, in particular to my team in the lab: Roshan Dathathri and Loc Hoang for the stimulating discussions, for the sleepless nights that we were working together on paper submission deadlines and for all the fun we had in the lab and at conferences. I am grateful to my senior colleagues in the lab, Andrew Lenharth, Donald Nguyen, Sreepathi Pai, Muhammad Amber Hassaan, and Rashid Kaleem for their mentorship.

I would like to thank Dr. Ramesh Peri at Intel for his support, guidance

and providing resources for my research. A special thanks to my team at Microsoft Research: Dr. Madan Musuvathi, Dr. Todd Mytkowicz, Dr. Saeed Maleki, Dr. Olli Saarikivi for amazing internship experience and exciting research opportunities.

Last but definitely not the least, I am extremely grateful to my parents, Devinder Kaur Gill and Bakhshish Singh Gill for their love, prayers, and sacrifices for educating and preparing me for my future. I am very thankful to my amazing wife, Anvita Seth, for her love, understanding and untiring support throughout my Ph.D. I thank my sister, Parminder Gill, all my cousin, and my in-laws for all the support and love without which this journey was not possible.

Compiler and System for Resilient Distributed Heterogeneous Graph Analytics

Publication No. ____________________

Gurbinder Singh Gill, Ph.D.
The University of Texas at Austin, 2020

Supervisor: Keshav Pingali

Graph analytics systems are used in a wide variety of applications including health care, electronic circuit design, machine learning, and cybersecurity. Graph analytics systems must handle very large graphs such as the Facebook friends graph, which has more than a billion nodes and 200 billion edges. Since machines have limited main memory, distributed-memory clusters with sufficient memory and computation power are required for processing of these graphs. In distributed graph analytics, the graph is partitioned among the machines in a cluster, and communication between partitions is implemented using a substrate like MPI. However, programming distributed-memory systems are not easy and the recent trend towards the processor heterogeneity has added to this complexity. To simplify the programming of graph applications on such platforms, this dissertation first presents a compiler called Abelian that

translates shared-memory descriptions of graph algorithms written in the Galois programming model into efficient code for distributed-memory platforms with heterogeneous processors.

An important runtime parameter to the compiler-generated distributed code is the partitioning policy. We present an experimental study of partitioning strategies for distributed work-efficient graph analytics applications on different CPU architecture clusters at large scale (up to 256 machines). Based on the study we present a simple rule of thumb to select among myriad policies.

Another challenge of distributed graph analytics that we address in this dissertation is to deal with machine fail-stop failures, which is an important concern especially for long-running graph analytics applications on large clusters. We present a novel communication and synchronization substrate called Phoenix that leverages the algorithmic properties of graph analytics applications to recover from faults with zero overheads during fault-free execution and show that Phoenix is 24x faster than previous state-of-the-art systems.

In this dissertation, we also look at the new opportunities for graph analytics on massive datasets brought by a new kind of byte-addressable memory technology with higher density and lower cost than DRAM such as intel Optane DC Persistent Memory. This enables the design of affordable systems that support up to 6TB of randomly accessible memory. In this dissertation, we present key runtime and algorithmic principles to consider when performing graph analytics on massive datasets on Optane DC Persistent Memory as well as highlight ideas that apply to graph analytics on all large-memory

platforms.

Finally, we show that our distributed graph analytics infrastructure can be used for a new domain of applications, in particular, embedding algorithms such as Word2Vec. Word2Vec trains the vector representations of words (also known as word embeddings) on large text corpus and resulting vector embeddings have been shown to capture semantic and syntactic relationships among words. Other examples include Node2Vec, Code2Vec, Sequence2Vec, etc (collectively known as Any2Vec) with a wide variety of uses. We formulate the training of such applications as a graph problem and present GraphAny2Vec, a distributed Any2Vec training framework that leverages the state-of-the-art distributed heterogeneous graph analytics infrastructure developed in this dissertation to scale Any2Vec training to large distributed clusters. GraphAny2Vec also demonstrates a novel way of combining model gradients during training, which allows it to scale without losing accuracy.

Table of Contents

Acknowledgments **v**

Abstract **vii**

List of Tables **xv**

List of Figures **xvii**

Chapter 1. Introduction **1**

1.1 Distributed Graph Analytics 4

1.1.1 Programming Model 4

1.1.2 Distributed-Memory Execution 5

1.2 Overview of Topics 5

1.2.1 Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory 5

1.2.2 Abelian: A Compiler for Graph Analytics on Distributed, Heterogeneous Platforms 6

1.2.3 A Study of Partitioning Policies for Graph Analytics on Large-scale Distributed Platforms 7

1.2.4 Phoenix: A Substrate for Resilient Distributed Graph Analytics 7

1.2.5 Distributed Training of Embeddings using Graph Analytics 8

1.3 Organization 8

Chapter 2. **10**

2.1 Motivation 10

2.2 Contributions 12

2.3 Optane PMM 14

2.4 Platforms and Graph Analytics Systems 17

2.5 Memory Hierarchy Issues 20

2.5.1 NUMA-aware Allocation . . . 20
2.5.2 NUMA-aware Migration . . . 24
2.5.3 Page Size Selection . . . 28
2.5.4 Summary . . . 30
2.6 Efficient Algorithms for Massive Graphs . . . 30
2.6.1 Classification of Graph Analytics Algorithms . . . 33
2.6.2 Algorithms for Very Large Graphs . . . 35
2.7 Evaluation of Graph Frameworks . . . 38
2.7.1 Galois, GAP and GraphIt on Optane PMM . . . 40
2.7.2 Medium-size graphs: Using Optane PMM vs. DDR4 DRAM . . . 43
2.7.3 Very large graphs: Using Optane PMM vs. a Cluster . . . 44
2.7.4 Out-of-core GridGraph in App-direct Mode vs. Galois in Memory Mode . . . 48
2.7.5 Discussion and Summary . . . 50
2.8 Related Work . . . 50

Chapter 3. **54**
3.1 Motivation . . . 54
3.2 Programming Model . . . 55
3.2.1 Implementation: . . . 57
3.2.2 Restrictions on operators . . . 59
3.3 Abelian Compiler . . . 59
3.3.1 Graph-data Access Analysis . . . 61
3.3.2 Restructuring computation . . . 62
3.3.2.1 Splitting operators . . . 63
3.3.2.2 Eliminating work-lists . . . 64
3.3.3 Inserting communication . . . 65
3.3.3.1 Fine-grained communication . . . 66
3.3.3.2 On-demand communication . . . 66
3.3.4 Device-specific compilers . . . 67
3.4 Experimental Evaluation . . . 67
3.4.1 Comparison with the state-of-the-art . . . 69

3.4.2 Impact of communication optimizations 71
3.5 Related Work 75

Chapter 4. **78**
4.1 Motivation 78
4.2 Background 79
4.3 Contributions 81
4.3.1 Lessons: 82
4.4 Partitioning Policies 83
4.4.1 1D Partitions 84
4.4.2 2D Block Partitions 85
4.4.3 Unrestricted Vertex-Cut Partitions 90
4.4.4 Discussion 90
4.5 Performance Model for communication 91
4.5.1 Replication Factor 92
4.5.2 Estimating Communication Volume 93
4.5.2.1 Edge-Cut (EC) 94
4.5.2.2 Hybrid Vertex-Cut (HVC) 94
4.5.2.3 Cartesian Vertex-Cut (CVC) 95
4.5.3 Micro-benchmarking to Estimate Communication Time 96
4.5.4 Discussion 97
4.6 Experimental Evaluation 99
4.6.1 Implementation 101
4.6.2 Partitioning Time 105
4.6.3 Load Balance 107
4.6.4 Execution Time 109
4.6.5 Strong Scaling 113
4.6.6 Communication Analysis 117
4.7 Choosing a Partitioning Policy 120
4.8 Related Work 124

Chapter 5. **128**
5.1 Motivation . . . 128
5.2 Contributions . . . 130
5.3 Background . . . 131
5.3.1 Fault Tolerant Distributed Graph Analytics . . . 131
5.4 Classes of Graph Analytics Algorithms . . . 135
5.4.1 Overview . . . 136
5.4.2 Classification of Graph Algorithms . . . 139
5.4.3 Distributed Graph Analytics . . . 141
5.5 Fault Tolerance in Distributed Memory . . . 142
5.5.1 Overview . . . 143
5.5.2 Self-Stabilizing Graph Algorithms . . . 146
5.5.3 Locally-Correcting Graph Algorithms . . . 147
5.5.4 Globally-Correcting Graph Algorithms . . . 150
5.5.5 Phoenix API . . . 153
5.6 Experimental Evaluation . . . 154
5.6.1 Experimental Setup . . . 155
5.6.2 Fault-free performance . . . 158
5.6.3 Overhead of fault tolerance . . . 162
5.7 Related Work . . . 165

Chapter 6. **168**
6.1 Motivation . . . 168
6.2 Contributions . . . 169
6.3 Background . . . 171
6.3.1 Stochastic Gradient Descent . . . 171
6.3.2 Training Any2Vec Embeddings . . . 174
6.3.3 Graph Analytics . . . 175
6.4 Distributed Any2Vec . . . 176
6.4.1 Overview of Distributed GraphAny2Vec . . . 176
6.4.2 Graph Generation and Partitioning . . . 179
6.4.3 Model Synchronization . . . 181

6.4.4 Communication Optimizations 183
6.5 Gradient Combiner 184
6.6 Proof of Properties of Gradient Combiner 189
6.6.1 Gradient Combiner (GC) Convergence Proof 190
6.7 Experimental Evaluation 193
6.7.1 Experimental Methodology 194
6.7.2 Comparing With The State-of-The-Art 199
6.7.3 Impact of Gradient Combiner (GC) 204
6.7.4 Impact of Communication Optimizations 207
6.8 Related Work 208

Chapter 7. Conclusion **211**

Bibliography **214**

List of Tables

2.1 Bandwidth (GB/s) of Intel Optane PMM. 17
2.2 Latency (ns) of Intel Optane PMM. 17
2.3 Inputs and their key properties. 18
2.4 Execution time (sec) of benchmarks in Galois on Optane PMM (OB) machine using efficient algorithms (non-vertex, asynchronous) and D-Galois on Stampede cluster (DM) using vertex programs with minimum number of hosts that hold the graph (5 hosts for clueweb12, and uk14, and 20 hosts for wdc12). Speedup of Optane PMM over Stampede cluster. Best times are highlighted in green. 44
2.5 Execution time (sec) of benchmarks in Galois on Optane PMM in Memory Mode (MM) and the state-of-the-art out-of-core graph analytics framework GridGraph on Optane PMM in App-direct Mode (AD). A 512 by 512 partition grid was used for GridGraph. Best times are highlighted in green. "—" indicates that system failed to finish in 2 hours. 49

3.1 Inputs and their key properties 68
3.2 Cluster configurations 69
3.3 Bridges: execution time (in seconds) on 16 GPUs for rmat28 . 69
3.4 Stampede: execution time (in seconds) (H: hosts) 70

4.1 Inputs and their properties. 98
4.2 Execution time of Gemini and D-Galois with EC. 98
4.3 Different initial Edge-Cut policies used for different benchmarks, inputs, and partitioning policies: IE: Incoming Edge-Cut, OE: Outgoing Edge-Cut, UE: Undirected Edge-Cut. 100
4.4 Graph partitioning time (includes time to load and construct graph) and static load balance of edges assigned to hosts on 256 KNL hosts. 103
4.5 Dynamic load balance: maximum-by-mean computation time on 256 KNL hosts. 105

4.6 Execution time (sec) for different partitioning policies, benchmarks, and inputs on KNL hosts. 107
4.7 Execution statistics of wdc12 on 256 KNL hosts. 108
4.8 Execution time (sec) on Skylake hosts for EC and CVC. . . . 109
4.9 Communication volume estimated by the model vs. observed communication volume on 128 KNL hosts for kron30. 119
4.10 % difference in execution time (excluding partitioning time) on KNL hosts between the partitioning strategy chosen by the decision tree and the optimal one (wdc12 is omitted because chosen one is always optimal; 0% means that chosen one is optimal). . 122
4.11 % difference in execution time (excluding partitioning time) on Skylake hosts between the partitioning strategy chosen by the decision tree and the optimal one. 123

5.1 Fault-free execution on 32 hosts of Wrangler. 133
5.2 Configuration of clusters. 155
5.3 Inputs and their key properties. 155
5.4 Fault-free execution of Phoenix and CR (R stands for the number of BSP-style rounds). 156

6.1 Datasets and their properties. 197
6.2 Word2Vec training time (hours) on a single host. 197
6.3 Word2Vec accuracy (semantic, syntactic, and total) of different frameworks relative to W2V. 198
6.4 Vertex2Vec training time (sec) of DeepWalk on 1 host vs. GraphVertex2Vec (GV2V) on 16 hosts. 201
6.5 Vertex2Vec accuracy (Macro F1 and Micro F1) of GV2V on 16 hosts relative to DeepWalk on 1 host. 201

List of Figures

2.1 Memory hierarchy of our 2 socket machine with 384GB of DRAM and 6TB of Intel Optane PMM. 14

2.2 Modes in Optane PMM. 16

2.3 Illustration of 3 different NUMA allocation policies on a 4-socket system: each policy distributes blocks (size B, which is the size of a page) of allocated memory among sockets differently . . . 21

2.4 Time to write memory allocated on Optane PMM and DDR4 DRAM using a micro-benchmark. 21

2.5 Execution time of bfs in Galois using small (4KB) and huge (2MB) page sizes with and without NUMA migration. 26

2.6 Breakdown of execution time of bfs in Galois using different page sizes for kron30 (left) and clueweb12 (right). 29

2.7 Execution time of different data-driven algorithms in Galois on Optane PMM using 96 threads. 31

2.8 Execution time of different data-driven algorithms in Galois on Entropy (1.5TB DDR4 DRAM) using 56 threads. 32

2.9 Execution time of benchmarks in GraphIt, GAP, GBBS, and Galois on Optane PMM using 96 threads. 36

2.10 Execution time of benchmarks in GBBS and Galois for wdc12 on Optane PMM using 96 threads. 37

2.11 Strong scaling in execution time of benchmarks in Galois using DDR4 DRAM and Optane PMM. 39

2.12 Execution time of benchmarks in Galois on Optane PMM machine and D-Galois on Stampede cluster with different configurations :- DB: Distributed Best (all threads on 256 hosts), DM: Distributed Min (all threads on min #hosts that hold graph), DS: Distributed Same (total 80 threads on min #hosts that hold graph), OS: Optane Same (same algorithm and threads as DS), OA: Optane All (same algorithm as DS, DM, and DB on 96 threads), OB: Optane Best (best algorithm on 96 threads). 46

3.1 Pagerank source program 58

3.2 Compiler-generated synchronization structures for field contrib in pagerank 58
3.3 Compiler-generated pagerank program 60
3.4 System Overview 61
3.5 32 KNL hosts on Stampede: clueweb12 and kron30. Different variants are: UnOpt (UO), Fine-Grained opt (FG), Fine-grained+On-demand opt (FO), Hand-Tuned(HT) 73
3.6 16 GPU devices on Bridges: rmat28 74
3.7 32 KNL hosts on Stampede: partitionings for bc on clueweb12 75

4.1 An example of partitioning a graph for 4 hosts (hosts depicted by different colors). 86
4.2 2D partitioning policies. 87
4.3 Communication patterns in BVC and CVC policies: red arrows are reductions and blue arrows are broadcasts for host 4. . . . 88
4.4 Performance of different communication patterns on different number of hosts on Stampede for different CPU architectures. 92
4.5 Communication volume of each computation round on clueweb12. 93
4.6 Execution time of D-Galois with different partitioning policies on KNL hosts. 111
4.7 Compute time of D-Galois with different partitioning policies on KNL hosts. 112
4.8 Replication factor of D-Galois with different partitioning policies on KNL hosts. 114
4.9 Communication volume of D-Galois with different partitioning policies on KNL hosts. 115
4.10 Breakdown of execution time on 256 hosts: kron30. 116
4.11 Breakdown of execution time on 256 hosts: clueweb12. 116
4.12 Breakdown of execution time on 256 hosts: wdc12 (XEC and BVC run out-of-memory). 117
4.13 Decision tree to choose a partitioning policy. 121

5.1 An example of partitioning a graph for two hosts in two different ways. 131
5.2 States during algorithm execution and recovery. 138
5.3 States of the graph in Figure 5.1(a), treated as an undirected graph, during the execution of greedy graph coloring. 144

5.4 Breadth first search (locally-correcting). 147
5.5 State of the graph in Figure 5.1(a) during the execution of data-driven breadth first search (locally-correcting). 148
5.6 State of the graph in Figure 5.1(a), treated as undirected, during the execution of data-driven k-core ($k = 4$). 151
5.7 Overheads in total time (%) of different fault scenarios with Phoenix and CR over fault-free execution. 157
5.8 Execution time (s) of Phoenix (Ph) and CR with different fault scenarios. 159
5.9 Overheads of different fault scenarios with Phoenix and CR over fault-free execution. 160
5.10 Execution time (s) of Phoenix (Ph) and CR with different fault scenarios for cf. 161
5.11 Execution time (s) of Phoenix (Ph), CR, and Checkpointing with Phoenix (CPh): 64 faults for wdc12. 161

6.1 Viewing Word2Vec in Skip-gram model as a graph. 173
6.2 Synchronization of the Skip-gram Model. 181
6.3 Synchronization of proxies in graph partitions. 182
6.4 Speedup of DMTK and GW2V on 32 hosts over W2V on 1 host. 202
6.5 Breakdown of training time of DMTK(GC) and GW2V(GC) on 32 hosts. 203
6.6 Strong scaling of GW2V (RMN: RepModel-Naive, RMO: RepModel-Opt, PMB: PullModel-Base, PMO: PullModel-Opt). 203
6.7 Accuracy of GW2V after each epoch for 1-billion dataset on 1 host (SM) and on 32 hosts using GC and AVG. 204
6.8 GW2V for 1-billion dataset on 32 hosts: % of gradients of a node that are orthogonal to each other in each epoch. 205
6.9 Breakdown of training time of GW2V with different communication schemes on 32 hosts. 206

Chapter 1

Introduction

Graph analytics systems are used in a wide variety of applications ranging from ranking webpages using algorithms like PageRank, finding the most influential people in social network graphs using betweenness-centrality, or finding shortest routes in maps using algorithms like single-source shortest path, etc. The other new emerging areas which are finding uses for graph analytics include health care, electronic circuit design, and cybersecurity. Graph sizes are rapidly increasing. Graph analytics systems must handle very large graphs such as the Facebook friends graph, which has more than a billion nodes and 200 billion edges, or the indexable Web graph, which has roughly 100 billion nodes and trillions of edges. Parallel computing is essential for processing graphs of this size in a reasonable time. The shared-memory graph analytics systems like Galois and Ligra process medium-scale graphs efficiently, but these systems cannot be used for graphs with billions of nodes and edges because of the limited main memory as well as processor count even on large servers.

In order to break free from the limited amount of main memory present on a single machine for shared-memory graph analytics, new kinds of byte-

addressable memory technology with higher density and lower cost than DRAM (such as Intel Optane DC Persistent Memory) can be used. This enables the design of affordable systems that support up to 6TB of randomly accessible memory. In this work, we present key runtime and algorithmic principles to consider when performing graph analytics on extreme-scale graphs on Optane DC Persistent Memory as well as highlight principles that can apply to graph analytics on all large-memory platforms.

In order to provide more compute resources, distributed-memory clusters with sufficient memory and computational power are required for processing large graphs. In distributed graph analytics, the graph is partitioned among the machines in a cluster, and communication between partitions is implemented using a substrate like MPI. However, programming distributed-memory systems are not easy and the recent trend towards processor heterogeneity has added to this complexity. To simplify the programming of graph applications on such platforms, we present a compiler called Abelian that translates shared-memory descriptions of graph algorithms written in the Galois programming model into efficient code for distributed-memory platforms with heterogeneous processors.

An important runtime parameter to the compiler-generated distributed code is the partitioning policy, which is left for the user to choose. We have observed that the application performance is sensitive to the partitioning policy chosen. However, there is no clear way to select the best partitioning policy for a given application and input, especially for the state-of-the-art distributed

work-efficient algorithms. In this work, we also present an experimental study of partitioning strategies for distributed work-efficient graph analytics applications on different CPU architecture clusters at large scale (up to 256 machines). Based on the study we present a simple rule of thumb to select among myriad policies.

Another challenge of distributed graph analytics that we address in this work is to deal with machine fail-stop failures, which is an important concern especially for long-running graph analytics applications on large clusters. We present a novel communication and synchronization substrate called Phoenix that leverages the algorithmic properties of graph analytics applications to recover from faults with zero overheads during fault-free execution and show that Phoenix is $\sim 24\times$ faster than state-of-the-art system (GraphX) and it outperforms the traditional checkpoint-restart techniques.

Last but not the least, we show that systems and infrastructure developed for graph analytics can also be used for a new domain of applications, in particular, the popular family of machine learning algorithms for unsupervised training of dense vector representations of entities known as Any2Vec. Word2Vec is a well-known example of Any2Vec, which trains vector representations of words (also known as word embeddings) on large text corpus and resulting vectors have been shown to capture semantic and syntactic relationships among words. Other examples include Node2Vec, Code2Vec, Sequence2Vec, etc. (collectively known as Any2Vec) with a wide variety of uses. We formulate the training of such applications as a graph problem and present

GraphAny2Vec, a distributed Any2Vec training framework that leverages the state-of-the-art distributed heterogeneous graph analytics infrastructure developed in this dissertation to scale Any2Vec training to large distributed clusters. GraphAny2Vec also demonstrates a novel way of combining model gradients during training, which allows it to scale without losing accuracy.

1.1 Distributed Graph Analytics

Section 1.1.1 and Section 1.1.2 outline the general details of the programming model and distributed execution model used in this work respectively. Any specific changes relevant to projects will be mentioned in their respective chapters.

1.1.1 Programming Model

Like other distributed graph analytics systems [70, 106, 195], this work uses a variant of the vertex programming model. Each node has one or more labels that are updated by an *operator* that reads the labels of the node and its neighbors, performs computation, and updates the labels of some of these nodes. Edges may also have labels that are read by the operator. Operators are generally categorized as *push-style* or *pull-style* operators. Pull-style operators update the label of the node to which the operator is applied while push-style operators update the labels of the neighbors. We also support updating label(s) of the node as well as label(s) of it's neighbors simultaneously within a single operator.

1.1.2 Distributed-Memory Execution

Distributed-memory graph analytics performs both computation and communication. The graph is partitioned between hosts at the start of the computation. Execution is done in rounds: in each round, a host applies the operator to graph nodes in its own partition and then participates in a global communication phase in which it exchanges information about labels of nodes at partition boundaries with other hosts. Since fine-grain communication is very expensive on current systems, execution models with coarse-grain communication, such as bulk-synchronous parallel (BSP) execution, are preferred [165].

1.2 Overview of Topics

Here is an overview of distributed graph analytics systems we developed:

1.2.1 Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory

Chapter 2 investigates Intel Optane DC Persistent Memory (Optane PMM), a new kind of byte-addressable memory with higher density and lower cost than DRAM. This enables the design of affordable systems that support up to 6TB of randomly accessible memory. In this chapter, we present key runtime and algorithmic principles to consider when performing graph analytics on extreme-scale graphs on Optane PMM as well as highlight principles

that can apply to graph analytics on all large-memory platforms. We evaluate four existing shared-memory graph frameworks and one outof-core graph framework on large real-world web-crawls using a machine with 6TB of Optane PMM. Our results show that frameworks using the runtime and algorithmic principles advocated in this chapter (i) perform significantly better than the others and (ii) are competitive with graph analytics frameworks running on large production clusters.

1.2.2 Abelian: A Compiler for Graph Analytics on Distributed, Heterogeneous Platforms

Chapter 3 presents Abelian [63], a compiler to simplify complexities of the programming model due to processor heterogeneity and distributed-memory in graph analytics. Abelian translates shared-memory descriptions of graph algorithms written in the Galois [118] programming model into efficient code for distributed-memory platforms with heterogeneous processors.

In our results we show that code produced by Abelian compiler is able to match the performance of handwritten distributed CPU and GPU programs as well as give 2.4× speedup over the state-of-the-art distributed CPU-only system (Gemini [195]), while boosting the programming productivity to write distributed graph analytics applications.

1.2.3 A Study of Partitioning Policies for Graph Analytics on Large-scale Distributed Platforms

In Chapter 4, we give an overview of our new way of thinking about graph partitioning as well as introduce various partitioning policies evaluated in the experimental study [66]. We present our findings on 2 different CPU architecture (KNL and Skylake) clusters with up to 256 machines using the Gluon [51] communication runtime which implements partitioning-specific optimizations for the state-of-the-art distributed work-efficient algorithms.

Our results show that although simple 1D partitioning strategies like Edge-Cuts perform well on a small number of machines, an alternative 2D partitioning strategy called Cartesian Vertex-Cut (CVC) performs better at scale even though paradoxically it has a higher replication factor and performs more communication than Edge-Cut partitioning does. Based on the study we also present a simple rule of thumb to select the best policy among myriad policies.

1.2.4 Phoenix: A Substrate for Resilient Distributed Graph Analytics

Chapter 5 presents Phoenix [52], a communication and synchronization substrate that implements a novel protocol for recovering from fail-stop faults when executing graph analytics applications on distributed-memory machines. The standard recovery technique in this space is checkpointing, which rolls back the state of the entire computation to a state that existed before the fault occurred. The insight behind Phoenix is that this is not necessary since

it is sufficient to continue the computation from a state that will ultimately produce the correct result.

Our evaluation shows that in the absence of faults, Phoenix is ∼24× faster than GraphX [181], which provides fault tolerance using the Spark system as well outperforms the traditional checkpoint-restart technique implemented in D-Galois [51].

1.2.5 Distributed Training of Embeddings using Graph Analytics

Chapter 6 presents GraphAny2Vec, a distributed training framework for a popular family of machine learning applications for unsupervised training of dense vector representations of entities known as Any2Vec. GraphAny2Vec exploits the fact that training of Any2Vec applications can be formulated as graph problems as shown in this chapter. GraphAny2Vec, built on top of our distributed heterogeneous graph analytics infrastructure, scales Any2Vec training to large distributed clusters. GraphAny2Vec also demonstrates a novel way of combining model gradients to honor data dependencies in SGD, which helps it scale without giving up convergence. It shows linear scalability up to 32 machines converging as fast as a sequential run in terms of epochs, thus reducing training time by 14×.

1.3 Organization

The document is organized as follows: Chapter 2 investigates Intel Optane DC Persistent Memory (Optane PMM), a new kind of byte-addressable

memory for graph analytics for massive datasets. Chapter 3 describes Abelian compiler to generate efficient distributed graph analytics programs from shared-memory specification. In Chapter 4, we describe our experimental evaluation of various partitioning policies for distributed graph analytics. Chapter 5 provides an overview of our novel resilient distributed graph analytics system. Finally Chapter 6 shows how graph analytics infrastructure can be used for a new class of machine leaning applications such as word2vec.

Chapter 2

Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory[1]

2.1 Motivation

Graph analytics systems must process graphs with tens of billions of nodes and trillions of edges. Since the main memory of most single machines is limited to a few hundred GBs, shared-memory graph analytics systems like Ligra [151], Galois [118], and GraphIt [190] cannot be used to perform in-memory processing of these large graphs. Two approaches have been used in the literature to circumvent this problem: (i) *out-of-core* processing and (ii) *distributed-memory* processing (described in Chapter 3, 4, and 5).

In out-of-core systems, the graph is stored in secondary storage (SS-D/disk), and portions of the graph are read into DRAM under software control for in-memory processing. State-of-the-art systems in this space include X-Stream [141], GridGraph [196], Mosaic [105], and BigSparse [87]. Secondary storage devices do not support random accesses efficiently, therefore, data

[1]This work has been accepted to be published in the proceedings of VLDB Endowment, 13(8), Tokyo, Japan, 2020 [64]. The key ideas introduced in this work as well as the experimental setup used were conceived by the first author while co-authors helped with its presentation.

must be fetched and written in blocks. As a consequence, algorithms that perform well on shared-memory machines often perform poorly in an out-of-core setting, and it is necessary to rethink algorithms and implementations when transitioning from in-memory graph processing to out-of-core processing. In addition, the graph may need to be preprocessed to organize the data into a layout that is friendly for out-of-core processing.

Large graphs can also be processed using distributed-memory clusters that have a sufficient number of machines and memory for in-memory processing of the graphs. The graph is partitioned among the machines in a cluster using one of many partitioning policies that have been studied in the literature [65] (Chapter 4). Communication is required during the computation to synchronize updates to node values across the entire cluster. State-of-the-art systems in this space include D-Galois [51] and Gemini [195]. Distributed-memory graph analytics systems have the advantage that they can be scaled out by adding new machines to provide additional memory and compute power. The overhead of communication can be reduced by choosing good partitioning policies, avoiding small messages, and optimizing metadata, but communication remains the bottleneck in these systems [51]. Obtaining access to large clusters may also be too expensive for many users.

Intel® Optane™ DC Persistent Memory (Optane PMM) is a new memory technology that promises to revolutionize this area. Optane PMM is byte-addressable memory which has the same form factor as DDR4 DRAM modules with higher memory density and lower cost. It has longer access times com-

pared to DRAM, but it is much faster than SSD. It can be set up to use the DRAM in the system as a very large cache. This allows a single machine to have up to 6TB of storage at relatively low cost, and in principle, it can be used to run memory-hungry applications without requiring the substantial reworking of algorithms and implementations that is required for out-of-core or distributed-memory processing.

2.2 Contributions

In this work, we explore the use of Optane PMM for analytics of very large graphs such as web-crawls up to 1TB in size. We design and present a set of studies conducted to determine the best practices for running graph analytics applications on Optane PMM (and large-memory systems in general). In particular, our studies make the following points:

1. NUMA-aware memory allocation of graph data structures that maximizes near-memory (DRAM treated as cache) usage is important on Optane PMM as cache misses on the platform are significantly slower than cache misses on DRAM. (Section 2.5)

2. Avoiding kernel overhead in managing pages while using Optane PMM is key to performance as kernel overhead on Optane PMM is higher due to higher access latency. (Section 2.5)

3. Algorithms must be designed to avoid high amounts of memory accesses on large memory systems: this means that graph frameworks should

give users the flexibility to write non-vertex, asynchronous programs by providing users with parallel data structures. (Section 2.6)

We evaluate four shared-memory graph analytics frameworks – Galois [118], GAP [15], GraphIt [190], and GBBS [56] – on Optane PMM to illustrate the importance of these practices for graph analytics on large-memory systems. We compare the performance of the best of these frameworks, Galois, with the state-of-the-art distributed graph analytics system, D-Galois, and show that a system running on Optane PMM that takes into account our best practices is competitive with the same algorithms run on D-Galois with up to 256 machines, and since the Optane PMM system supports more efficient shared-memory algorithms such as those using pointer-jumping which are difficult to implement on distributed-memory machines, applications using the more efficient algorithms can even outperform distributed-memory execution. We also evaluate the out-of-core graph analytics system GridGraph [196] using Optane PMM as external storage in app-direct mode (explained in Section 2.3) and show that using Optane PMM as main memory in memory mode (explained in Section 2.3) is orders of magnitude faster than app-direct as it allows OS to use DRAM as last-level cache and supports more sophisticated algorithms from current shared-memory graph analytics systems that out-of-core systems do not support (in particular, non-vertex programs and asynchronous data-driven algorithms).

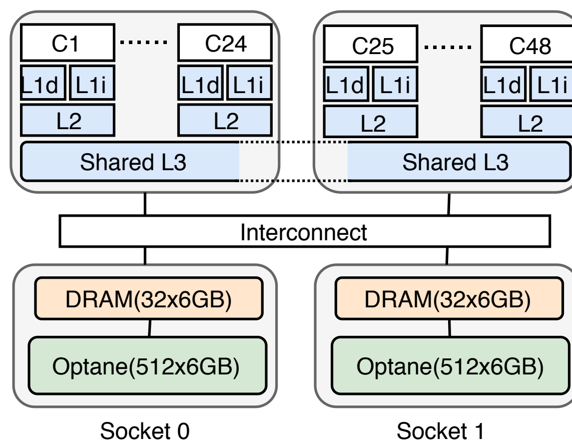


Figure 2.1: Memory hierarchy of our 2 socket machine with 384GB of DRAM and 6TB of Intel Optane PMM.

2.3 Optane PMM

Optane PMM is a new memory technology that delivers a combination of affordable large capacity and persistence (non-volatility). As shown in Figure 2.1, this memory adds one more level in the memory hierarchy. This memory comes in the same form factor as a DDR4 memory module and has the same electrical and physical interfaces. However, it uses a different protocol than DDR4 which means that the CPU must have Optane PMM support in its memory controller. Similar to the DRAM distribution in NUMA systems, the Optane PMM modules are also distributed among sockets. Figure 2.1 shows an example of a two socket machine with 6TB of Optane PMM split among the sockets. Optane PMM can be configured as volatile main memory (called memory mode), persistent memory (called app-direct mode), or a combination of both, as shown in Figure 2.2.

Memory Mode: In memory mode, the operating system treats Op-

tane PMM as main memory, and DRAM acts as direct-mapped (physically indexed and physically tagged) cache called *near-memory*. The Granularity of caching from Optane PMM to DRAM is 4KB (small page size). This enables the system to deliver DRAM-like performance at substantially lower cost and power with no modifications to the application. Although the memory media is persistent, the software sees it as volatile memory. This enables a common two-socket system to provide up to 6TB of randomly accessible storage, which is difficult and expensive to implement with DRAM.

Traditional code optimization techniques like blocking can be used to tune applications to run well in this configuration. In addition, software needs to be optimized for certain asymmetries in machines with Optane PMM. Optane PMM modules on a given socket can use only the DRAM present in its local NUMA node as near-memory. Therefore, in addition to NUMA allocation considerations, software using Optane PMM has to take the near-memory hit rate into account as well because the cost of a local near-memory miss is much higher than the remote near-memory hit (this is discussed in detail in Section 2.5). Therefore, it is beneficial to allocate memory so that the system can utilize more DRAM as near-memory even if it means more remote NUMA accesses.

App-direct Mode: In app-direct mode, Optane PMM modules are provisioned as byte-addressable persistent memory. The Persistent Memory Development Kit (PMDK) [5] is a library that can make programming in app-direct mode efficient. One compelling case for app-direct mode is

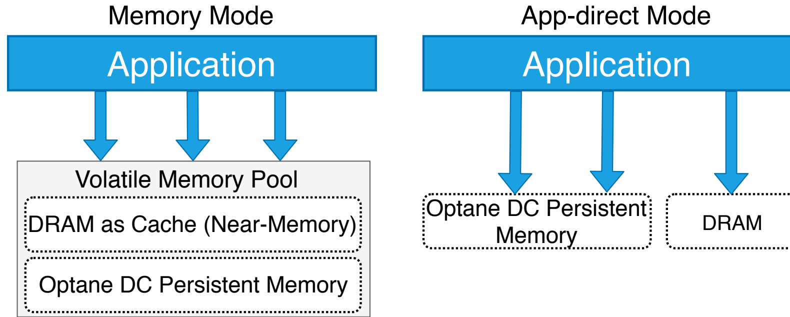


Figure 2.2: Modes in Optane PMM.

in large memory databases where indices can be stored in persistent memory to avoid rebuilding them on reboot, achieving a significant reduction in restart time. Optane PMM modules can be configured and managed using an API or a command line interface provided by the `ipmctl` [49] OS utility in Linux. For example, `ipmctl` can be used to configure the machine to use $x\%$ of Optane PMM modules capacity in the memory mode and the rest in the app-direct mode (`ipmctl create -goal MemoryMode=x PersistentMemoryType=AppD-irect`); for $x > 0$, all the DRAM present on the machine is used as the cache (`near-memory`). When all the Optane PMM modules are configured in app-direct mode using `ipmctl`, DRAM is used as the main volatile memory.

Detailed specifications for the particular machine with Optane PMM used in our study are given in Section 2.4. Tables 2.1 and 2.2 show the bandwidth and latency of PMM that we observe on our machine. Although Optane PMM is slower than DDR4, the large capacity offered by these DIMMs enables us to analyze much larger datasets on a single machine than was possible earlier. In this work, we focus mainly on memory mode; we use app-direct mode

Mode		Read		Write	
		Local	Remote	Local	Remote
Memory	Random	90.0	34.0	50.0	29.5
	Sequential	106.0	100.0	54.0	29.5
App-direct	Random	8.2	5.5	3.6	2.3
	Sequential	31.0	21.0	10.5	7.5

Table 2.1: Bandwidth (GB/s) of Intel Optane PMM.

Mode	Local	Remote
Memory	95.0	150.0
App-direct	164.0	232.0

Table 2.2: Latency (ns) of Intel Optane PMM.

for running the out-of-core graph analytics system GridGraph [196].

2.4 Platforms and Graph Analytics Systems

Optane PMM experiments were conducted on a 2 socket machine with Intel's second generation Xeon scalable processor ("Cascade Lake") with 48 cores (we use up to 96 threads with hyperthreading) with a clock rate of 2.2 Ghz. The machine has 6TB of Optane PMM, 384GB of DDR4 RAM, and 32KB L1, 1MB L2, and 33MB L3 data caches, as shown in Figure 2.1. The system has a 4-way associative data TLB with 64 entries for 2KB pages (referred to as small pages), 32 entries for 2MB pages (referred to as huge pages), and 4 entries for 1GB pages. Source code is compiled with g++ 7.3. We used the same machine for DRAM experiments by configuring the Optane PMM modules using `ipmctl` utility to run entirely in app-direct mode and use DRAM as the main volatile memory (which is equivalent to removing the Optane PMM modules). We also use Optane PMM modules in app-direct

	kron30	clueweb12	uk14	rmat32	wdc12
$\|V\|$	1,073M	978M	788M	4295M	3,563M
$\|E\|$	10,791M	42,574M	47,615M	68,719M	128,736M
$\|E\|/\|V\|$	16	44	60.4	16	36
max D_{out}	3.2M	7,447	16,365	10.4M	55,931
max D_{in}	3.2M	75M	8.6M	10.4M	95M
Est. diameter	6	498	2498	7	5274
Size (GB)	136	325	361	544	986

Table 2.3: Inputs and their key properties.

mode to run GridGraph, an out-of-core graph analytics system: DRAM is used as main memory and Optane PMM modules are used as external storage.

Transparent Huge Pages (THP) are enabled (default in Linux). To collect hardware counters and analyze performance, we used Intel's Vtune Amplifier [48] and Platform Profiler [47].

To show that our study of algorithms for massive graphs (Section 2.6) is independent of machine architecture, we also conducted experiments on a large DRAM 4 socket machine that we call Entropy. Entropy uses Intel Xeon Platinum 8176 ("Skylake") processors with a total of 112 cores with a clock rate of 2.2 Ghz, 1.5TB of DDR4 DRAM, and 32KB L1, 1MB L2, and 38MB L3 data caches. Source code is compiled with g++ 5.4. For our experiments on Entropy, we only used 56 threads, restricting our experiments to 2 sockets.

Distributed-memory experiments were conducted on Stampede2 [155] cluster at the Texas Advanced Computing Center using up to 256 Intel Xeon Platinum 8160 ("Skylake") 2 socket machines with 48 cores with a clock rate of 2.1 Ghz, 192GB DDR4 RAM, and 32KB L1, 1MB L2, and 33MB L3 data caches. The machines are connected with a 100Gb/s Intel Omni-Path inter-

connect. We use LCI [50] for message passing between machines. Source code is compiled with g++ 7.1.

Table 2.3 specifies the input graphs: clueweb12 [134], uk14 [18, 22], and wdc12 [110] are web-crawls (wdc12 is the largest publicly available one) used throughout our study. kron30 and rmat32 are randomized scale-free graphs generated using kron [102] and rmat [36] generators (using weights of 0.57, 0.19, 0.19, and 0.05, as suggested by graph500 [1]). kron30 and clueweb12 fit into DRAM, so we use them to illustrate differences in workloads that fit into DRAM and those that do not. The other graphs – uk14, rmat32, and wdc12 – do not fit in DRAM on our Optane PMM machine. We observe that uk14 and wdc12 have non-trivial diameters, whereas rmat32 has a very small diameter. We believe that rmat32 does not represent real-world datasets, so we exclude it in all our experiments except to show the impact of diameter in our study of algorithms (Section 2.6). All graphs are unweighted, so we generate random weights.

Our evaluation uses 7 benchmarks: single-source betweenness centrality (bc), breadth-first search (bfs), connected components (cc), k-core decomposition (kcore), pagerank (pr), single-source shortest path (sssp), and triangle counting (tc). The only benchmark that uses weights is sssp. The source node for bc, bfs, and sssp is the maximum out-degree node. The tolerance for pr is 10^{-6}. The k in kcore is 100. *All benchmarks are run until convergence except for pr, which is run for up to 100 rounds.* We present the mean of 3 runs for the main experiments.

The shared-memory graph analytics frameworks we use are Galois [118], GAP [15], GraphIt [190], GBBS [56], and D-Galois [51]. These frameworks are described in more detail in Section 5.6. D-Galois is a distributed-memory framework; the rest are shared-memory frameworks. GridGraph [196] is an out-of-core framework that streams graph topology and data into memory from external storage (in this case, Optane PMM in app-direct mode).

2.5 Memory Hierarchy Issues

This section shows that on Optane PMM machines, the overhead of memory operations such as NUMA memory accesses, handling cache misses, and page table maintenance are higher than on regular DRAM machines. For good performance, these overheads must be reduced by intelligent memory allocation and by reducing the time spent in the kernel for page-table maintenance. There are three main issues to address: NUMA-aware allocation (Section 2.5.1), NUMA-aware migration (Section 2.5.2), and page size selection (Section 2.5.3).

2.5.1 NUMA-aware Allocation

NUMA-aware allocation attempts to increase bandwidth and reduce latency of memory accesses by allocating memory on the same NUMA node as the cores that are likely to access that memory. Allocation policies fall into three main categories: (a) *NUMA local*, which allocates memory on a node specified at allocation time (if there is not enough memory available on

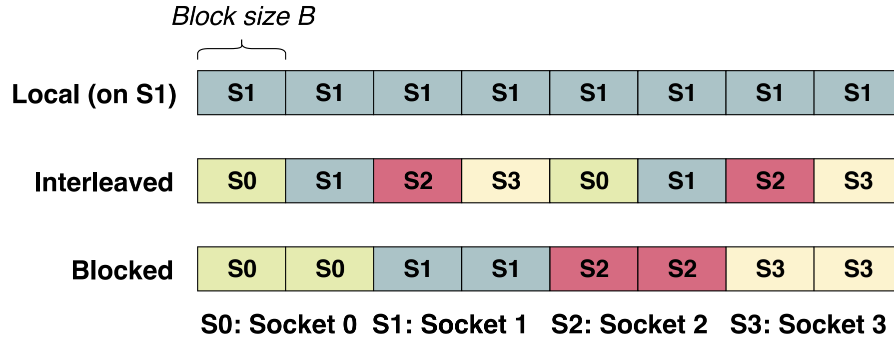
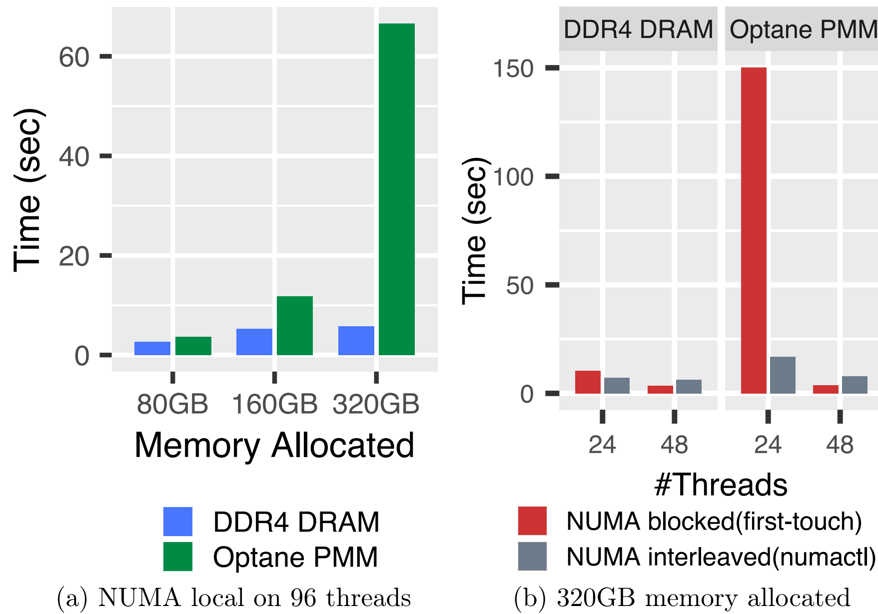


Figure 2.3: Illustration of 3 different NUMA allocation policies on a 4-socket system: each policy distributes blocks (size B, which is the size of a page) of allocated memory among sockets differently



(a) NUMA local on 96 threads (b) 320GB memory allocated

Figure 2.4: Time to write memory allocated on Optane PMM and DDR4 DRAM using a micro-benchmark.

the preferred node, other NUMA nodes will be used), (b) *NUMA interleaved*, which allocates the memory by interleaving physical pages across NUMA nodes in a round-robin fashion, and (c) *NUMA blocked*, which blocks the physical pages to be allocated and distributes the blocks among NUMA nodes on the system (illustrated in Figure 2.3).

There are several ways for application programs to specify the allocation policy. The policy can be set globally by using OS utilities such as `numactl` on Linux. To allow different policies to be used in different allocations, applications can use the OS-provided NUMA allocation library (`numa.h` in Linux), which contains a variety of `numa_alloc` functions. OS-based approaches, however, can only use the NUMA local or interleaved policies. Another way to get fine-grained NUMA-aware allocation is to manually allocate memory using `anonymous mmap` and have threads on different sockets inside the application touch the pages (called as `first-touch`) to allocate them on the desired NUMA nodes. This method, unlike OS-provided methods, allows applications to implement application-specific NUMA-aware allocation policies.

To understand the differences in local, interleaved, and blocked NUMA allocation policies on our Optane PMM setup, we use a simple micro-benchmark that allocates different amounts of memory using different NUMA allocation policies and writes to each location once using t threads where each thread gets a contiguous block to write sequentially. To explore the effects of NUMA on different platforms, we run this microbenchmark two machines: one with only DDR4 DRAM and one using Optane PMM. The following micro-benchmark

results show that on the Optane PMM machine, applications must not only maximize local NUMA accesses, but *must also use a NUMA policy that maximizes the amount of near-memory used to reduce DRAM conflict misses* (recall that DRAM acts like a direct-mapped cache called near-memory for Optane PMM in this mode).

NUMA Local. Figure 2.4(a) shows the execution time of the microbenchmark on DDR4 DRAM and Optane PMM for the NUMA local allocation policy using $t = 96$ and different allocation amounts. Using NUMA local, all the memory of socket 0 is used before memory from socket 1 is allocated. We observe that going from 80GB to 160GB increases the execution time by 2× for both DRAM and Optane PMM: this is expected since we are increasing the work by 2×. Going from 160GB to 320GB also increases the work by 2×. For DRAM, a 320GB allocation spills to the other socket (each socket has only 192GB), and this increases the effective bandwidth by 2×, so the execution time does not change much. In Optane PMM , however, the 320GB is allocated entirely on socket 0 as our machine has 3TB per socket. Since there is no change in bandwidth, one would expect the performance to degrade by 2×, but it actually degrades by 5.6×. This is because the machine only gets to use 192GB of DRAM as near-memory cache; this cannot fit 320GB, so the conflict miss rate of the DRAM accesses increase by roughly 1.8×. This illustrates that (i) near-memory conflict misses are detrimental to the performance for Optane PMM and (ii) NUMA local policy is not suitable for allocations larger than 192GB on our setup.

NUMA Interleaved and Blocked. The execution times of the micro-benchmark on DDR4 DRAM and Optane PMM for the NUMA interleaved and blocked allocation policies using an allocation of 320GB and different thread counts is shown in Figure 2.4(b). For DRAM, both policies are similar for different t. When $t \leq 24$ on Optane PMM, NUMA blocked only allocates memory on socket 0 (because it uses `first-touch`), so performance degrades 39× compared to 48 thread execution as 320GB does not fit in the near-memory of a single socket. This illustrates that the cost of local near-memory misses is much higher than the cost of remote near-memory hits. In contrast, the NUMA interleaved policy for 24 threads uses both sockets and improves performance by 9× over NUMA blocked even though 50% of accesses are remote when $t \leq 24$. NUMA interleaved performs worse than blocked when $t = 48$, as both allocation policies are able to fit 320GB in the near-memory of 2 sockets (384GB); however, NUMA interleaved results in more remote accesses as compared to NUMA blocked.

2.5.2 NUMA-aware Migration

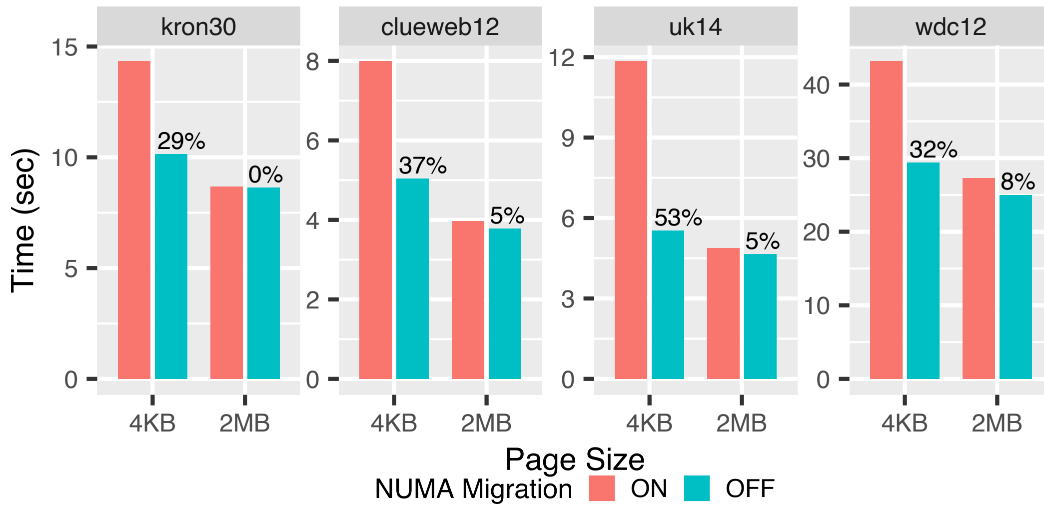
When an OS-level NUMA allocation policy is not specified (the application-level NUMA policy is not visible to the OS), the OS can dynamically migrate data among NUMA nodes to increase the proportion of local NUMA accesses. NUMA page migrations are helpful for multiple applications sharing a single system as they try to move pages closer to the cores assigned to each application. However, for a single application running on the machine, this

policy may not always be useful, especially when application has specified its own NUMA allocation policy using its knowledge of memory access patterns.

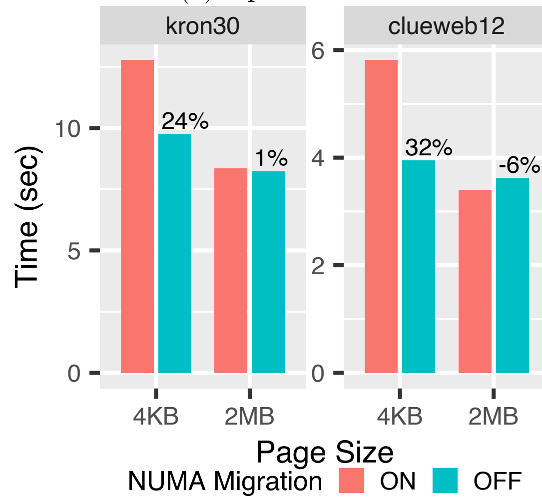
OS-directed migration has many overheads: (a) it requires book-keeping to track accesses to the pages to select pages for migration, and (b) migration changes the virtual-to-physical address mapping, which makes the Page Table Entries (PTEs) cached in CPU's Translation Lookaside Buffers (TLBs) stale, causing TLB shootdown on each core. TLB shootdown involves slow operations such as issuing inter-processor-interrupts (IPIs), and it also increases TLB misses.

Graph analytics applications tend to have irregular access patterns: accesses are arbitrary, so there may be many shared accesses across NUMA sockets. To examine the effects of page migration on graph analytics applications, we run breadth-first search (bfs) (similar trends are observed for other benchmarks) using Galois [118] using NUMA interleaved allocation for different input graphs on both Optane PMM and DDR4 DRAM with NUMA migration on and off. We also examine the effects of page migration for different page sizes: (a) 4KB small page and (b) 2MB huge page. The results of these experiments below suggest that *NUMA migration should be turned off for graph analytics applications on Optane PMM.*

Figure 2.5 shows the effect of NUMA migration where the number on each bar presents the % change in the execution time when NUMA migration is turned off. A positive number means turning migration off improves performance. Performance improves in most cases if NUMA migration is turned



(a) Optane PMM



(b) DDR4 DRAM

Figure 2.5: Execution time of bfs in Galois using small (4KB) and huge (2MB) page sizes with and without NUMA migration.

off. Figure 2.6 shows that the time spent in user code is not affected by the NUMA migrations, which shows that migrations are adding additional kernel time overhead without giving significant benefits. Another way to measure the efficacy of the page migrations is to measure the % of local near-memory (DRAM) accesses in Optane PMM: if migration is beneficial, then this should increase. However, this does not change by more than 1%. Figure 2.6 shows that NUMA migrations hurt performance more on Optane PMM as compared to DRAM as time spent in the kernel is higher. This is due to (a) higher cost of bookkeeping as memory accesses to kernel data structures are more expensive on Optane and (b) higher cost of TLB shootdown as it increases the access latency to the near-memory (DRAM) being used as a direct-mapped cache since TLB translation is on the critical path.[2]. Larger graphs exacerbate this effect as they use more pages.

4KB small page size shows more performance improvement than 2MB huge pages when turning page migrations off. We observe that the number of migrations is in the millions for small pages and in the hundreds for huge pages. The finer granularity of small pages makes them more prone to migrations, leading to more TLB shootdowns and data TLB misses. Therefore, for small pages, the number of data TLB misses reduces by $\sim 2\times$ by turning off the NUMA migrations for all the graphs. The number of small pages being $512\times$ the number of huge pages also increases the bookkeeping overhead in the OS.

[2]Since near-memory (DRAM) is used as physically indexed and physically tagged direct-mapped cache, virtual addresses need to be translated to physical addresses before cache (near-memory) can be accessed.

This is reflected in the amount of time spent in the OS kernel due to NUMA migrations, as seen in Figure 2.6. The time spent in the kernel is more for the smaller page size than for the larger page size if page migration is turned on.

2.5.3 Page Size Selection

When memory sizes and workload sizes grow, the time spent handling TLB misses can become a performance bottleneck since large working sets need many virtual-to-physical address translations that may not be cached in the TLB. This bottleneck can be tackled either (a) by increasing the TLB size in the hardware or (b) by increasing the page size. The TLB size is determined by the micro-architecture and cannot easily be changed by a user. On the other hand, processors allow users to customize page sizes as different page sizes may work best for different workloads. For example, x86 supports traditional 4KB small pages as well as 2MB and 1GB huge pages.

We studied the impact of page size on graph analytics using a 4KB small page size and a 2MB huge page size, which are supported by Linux. We did not include results for 1GB page size as it requires special setup; moreover, we do not expect to gain much from 1GB page size as the hardware supports fewer TLB entries for 1GB page size. We run bfs (similar behavior was observed for other benchmarks) using Galois [118] with no NUMA migration and NUMA interleaved memory allocation policy for various large graphs on (a) Optane PMM and (b) DDR4 DRAM. The results below suggest that suggest that *a page size of 2MB is good for graph analytics on Optane PMM.*

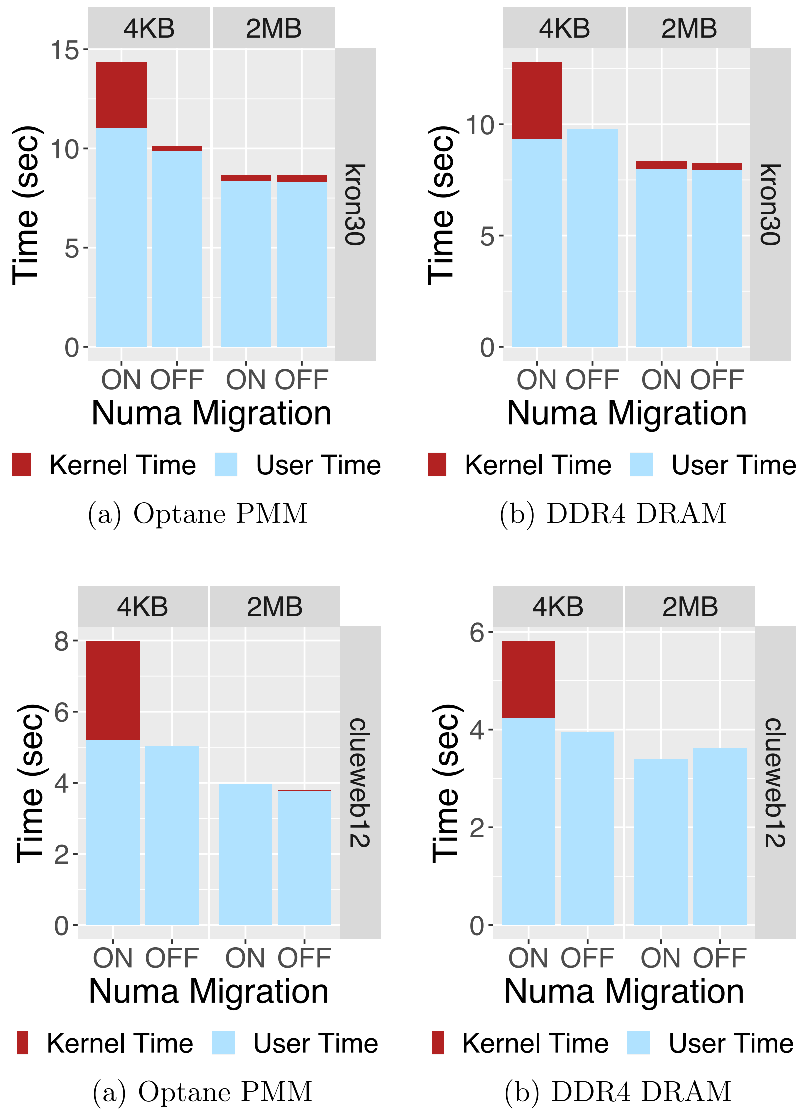


Figure 2.6: Breakdown of execution time of bfs in Galois using different page sizes for kron30 (left) and clueweb12 (right).

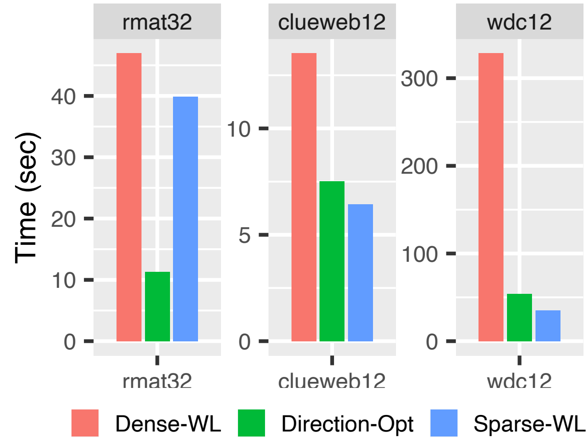
Figure 2.5 shows the results of bfs with various page sizes, and we observe that using huge pages is always beneficial on large graphs as huge pages reduce the number of pages required by 512×, which reduces the number of TLB misses (3.2× for clueweb12, 11.2× for uk14 and 1.9× for wdc12) and CPU cycles spent on page walking on TLB misses (7.3× for clueweb12, 12.5× for uk14 and 8.8× for wdc12). We also observe that the benefits of huge pages are higher on Optane PMM than on DRAM because TLB misses increase the near-memory access latency. Huge pages increase the TLB reach (TLB size × page size), thereby reducing the TLB misses.

2.5.4 Summary

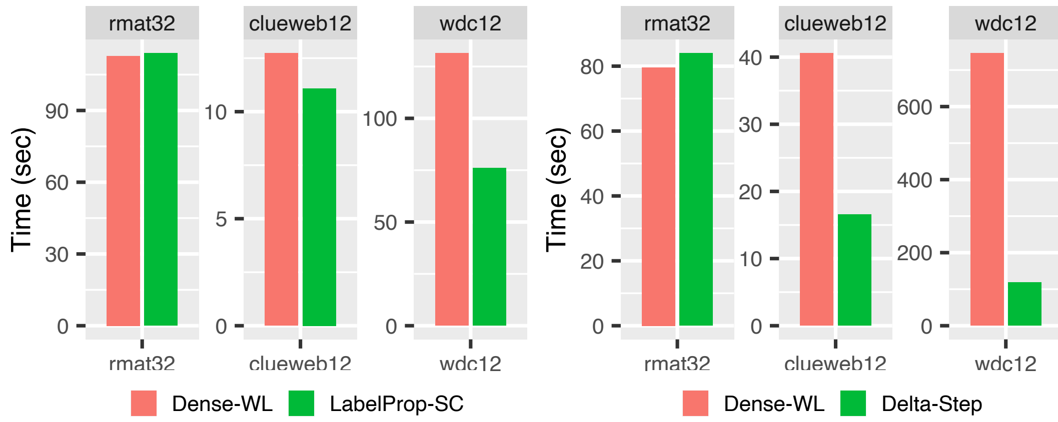
For high-performance graph analytics on the Optane PMM system, we recommend (i) using NUMA interleaved or blocked memory allocation rather than NUMA local, particularly for large allocations (> 192GB), (ii) turning off NUMA page migration, and (iii) using 2MB huge pages.

2.6 Efficient Algorithms for Massive Graphs

In general, there are many algorithms that can be used to solve a given graph analytics problem; for example, the single-source shortest-path (sssp) problem can be solved using Dijkstra's algorithm, the Bellman-Ford algorithm, chaotic relaxation, and delta-stepping. These algorithms may have different asymptotic complexities and different amounts of parallelism. For example for a graph $G = (V, E)$, the asymptotic complexity of Dijkstra's algorithm



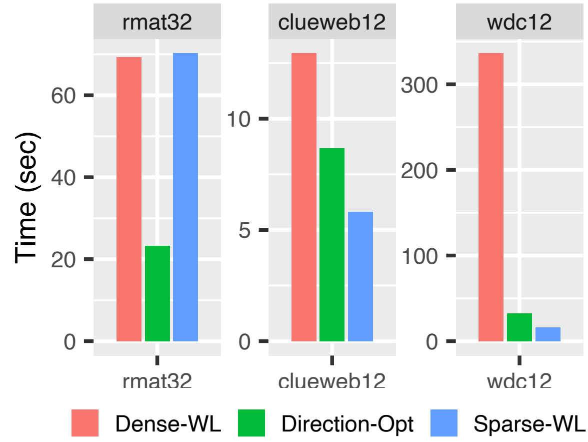
(a) bfs



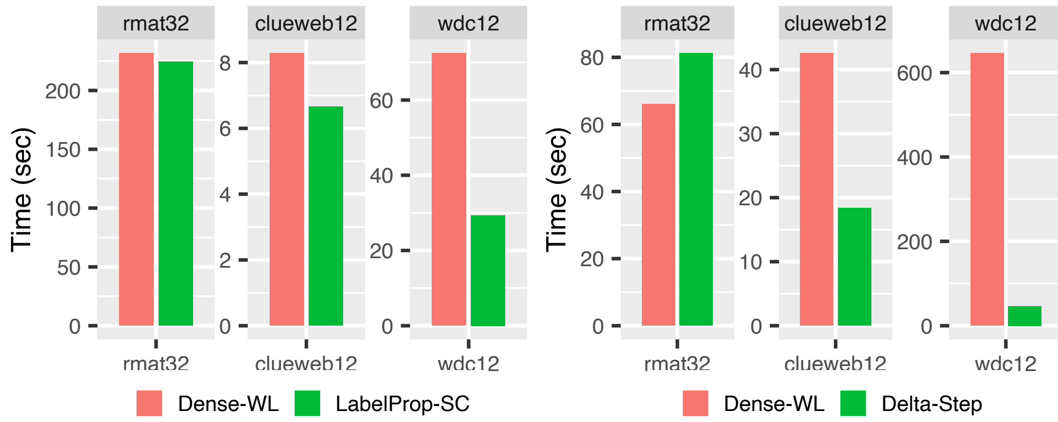
(b) cc

(c) sssp

Figure 2.7: Execution time of different data-driven algorithms in Galois on Optane PMM using 96 threads.



(a) bfs



(b) cc

(c) sssp

Figure 2.8: Execution time of different data-driven algorithms in Galois on Entropy (1.5TB DDR4 DRAM) using 56 threads.

is $O(|E|*log(|V|))$ while Bellman-Ford is $O(|E|*|V|)$, but for most graphs, Dijkstra's algorithm has little parallelism compared to Bellman-Ford. Complicating the picture further is the fact that a given algorithm can usually be implemented in different ways, and these implementation details may affect parallel performance dramatically; implementations that use fine-grain locking for example usually perform better than those that use coarse-grain locking.

This section presents a classification of graph analytics algorithms that is useful for understanding parallel performance [130]. We also present experimental results that provide insights into which classes of algorithms perform well on very large input graphs that can run on Optane PMM.

2.6.1 Classification of Graph Analytics Algorithms

Operators: In graph analytics algorithms, each vertex has one or more labels that are initialized at the start of the computation and then updated repeatedly during the computation until a quiescence condition is reached. Label updates are performed by applying an *operator* to *active vertices* in the graph. In some systems such as Galois [118], an operator may read and update an arbitrary portion of the graph surrounding the active vertex; this portion is called its *neighborhood*. Most shared-memory systems such as Ligra [56, 151] and GraphIt [190] only support a limited class of operators called *vertex operators* whose neighborhoods are only the immediate neighbors of the active vertex. A *push-style* operator updates the labels of the neighbors of the active vertex, while a *pull-style* operator updates the label of only the active vertex.

Direction-optimizing implementations [14] can switch between push and pull style operators dynamically, but they require a reverse edge for every forward edge in the graph, which doubles the memory footprint of the graph.

Schedule: To find active vertices in the graph, algorithms take one of two approaches. A *topology-driven* algorithm executes in rounds, and in each round, it applies the operator to all the graph vertices; Bellman-Ford sssp is an example. These algorithms are simple to implement, but they may not be work-efficient if there are few active vertices in a lot of rounds. To address this, *data-driven* algorithms track active vertices explicitly and apply the operator only to these vertices. At the start of the algorithm, some vertices are active; applying the operator to an active vertex may activate other vertices, and operator application continues until there are no active vertices in the graph. Dijkstra and delta-stepping sssp algorithms are examples. Active vertices can be tracked using a bit-vector of size V if there are V vertices in the graph: we call this a *dense worklist* [56, 151, 190]. Other implementations keep an explicit worklist of active vertices [118]: we call this a *sparse worklist*.

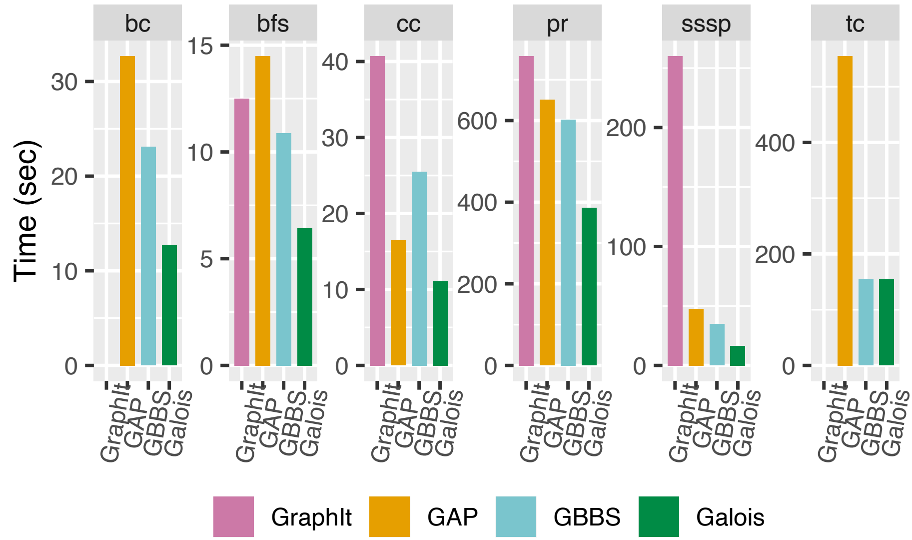
Some implementations of data-driven algorithms execute in *bulk-synchronous* rounds: they keep a *current* and a *next* worklist, and in each round, they process only the vertices in the current worklist and add activated vertices to the next worklist. The worklists can be dense or sparse. In contrast, *asynchronous* data-driven implementations have no notion of rounds; they maintain a single sparse worklist, pushing and popping active vertices from this worklist until it is empty.

2.6.2 Algorithms for Very Large Graphs

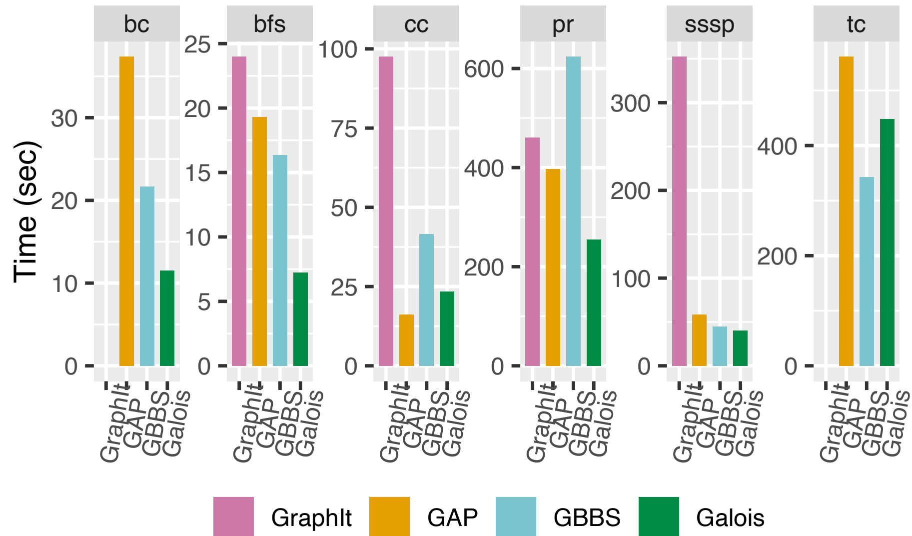
At present, very large graphs are analyzed using clusters or out-of-core systems, but the programs on these systems are restricted to vertex programs and round-based execution. This is not considered to be a serious limitation for power-law graphs since they have a small diameter, and information does not have to propagate many hops in these graphs. In fact, no graph analytics framework other than Galois provides sparse worklists, so they cannot support asynchronous data-driven algorithms and most of them are restricted to vertex programs.

Using the Optane PMM system, we were able to use a single machine to perform analytics on very large graphs, and our results suggest that conventional wisdom in this area needs to be revised. *The key issue is highlighted by Table 2.3: clueweb12, uk14, and wdc12, which are real-world web-crawls, actually have a very high diameter (shown in Table 2.3) compared to kron30 and rmat32, the synthetic power-law graphs.* We show below that for standard graph analytics problems, the best-performing algorithms for these graphs may be (a) non-vertex programs and (b) asynchronous data-driven algorithms, which require sparse worklists. This is because these algorithms have better work-efficiency and because they make fewer memory accesses, which is beneficial for performance especially on Optane PMM where memory accesses are more expensive.

Figure 2.7 shows the execution time of different data-driven algorithms for bfs, cc, and sssp on Optane PMM using the rmat32, clueweb12, and wdc12



(a) clueweb12



(b) uk14

Figure 2.9: Execution time of benchmarks in GraphIt, GAP, GBBS, and Galois on Optane PMM using 96 threads.

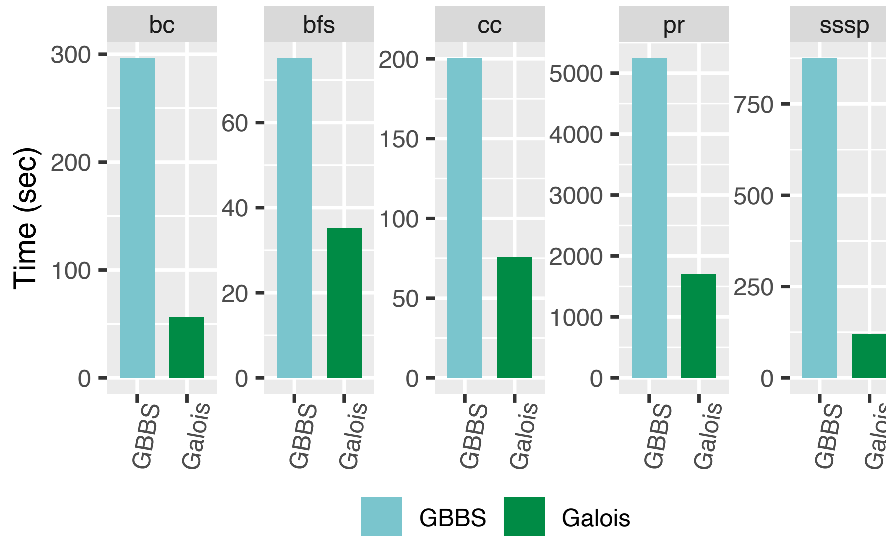


Figure 2.10: Execution time of benchmarks in GBBS and Galois for wdc12 on Optane PMM using 96 threads.

graphs on the Galois system. For bfs, all algorithms are bulk-synchronous. A vertex program with direction optimization (that uses dense worklists) performs well for rmat32 since it has a low-diameter, but for the real-world web-crawls, which have much higher diameter, it is outperformed by an implementation with a push-style operator and sparse worklists since this algorithm has a lower memory footprint, makes fewer memory accesses, and is more efficient in the later rounds when there are few active vertices. For cc, bulk-synchronous label propagation combined with short-cutting (LabelProp-SC) [156], which uses a non-vertex operator, is used. It is a variant of the Pointer-Jumping algorithm where after every round of label propagation it jumps one level unlike the Pointer-Jumping where it goes all way to the common descendent. LabelProp-SC exhibits better locality as compared Pointer-

Jumping and significantly outperforms the bulk-synchronous algorithm that uses a simple label propagation vertex operator for the real-world web-crawls. For sssp, the asynchronous delta-stepping algorithm, which maintains a sparse worklist, significantly outperforms the bulk-synchronous data-driven algorithm with dense worklists. These findings do not apply only to Optane PMM: Figure 2.8 shows the same experiments for bfs, sssp, and cc conducted on Entropy (DDR4 DRAM machine). The trends are similar to those on the machine with Optane PMM.

Summary: Large real-world web-crawls, which are the largest graphs available today, actually have a high diameter, unlike synthetically generated rmat and kron graphs. Therefore, conclusions drawn from experiments with rmat and kron graphs can be misleading. On current distributed-memory and out-of-core platforms, one is forced to use vertex programs, but on machines with Optane PMM, *it is advantageous to use algorithms with non-vertex operators and sparse worklists of active vertices that allow for asynchronous execution.* Frameworks that support only vertex operators or that do not have sparse worklists are at a disadvantage on this platform when processing large real-world web-crawls, as we show next.

2.7 Evaluation of Graph Frameworks

In this section, we evaluate several graph frameworks on Optane PMM in the context of the performance guidelines presented in Sections 2.5 and 2.6. In Section 2.7.1, four shared-memory graph analytics systems - Galois [118],

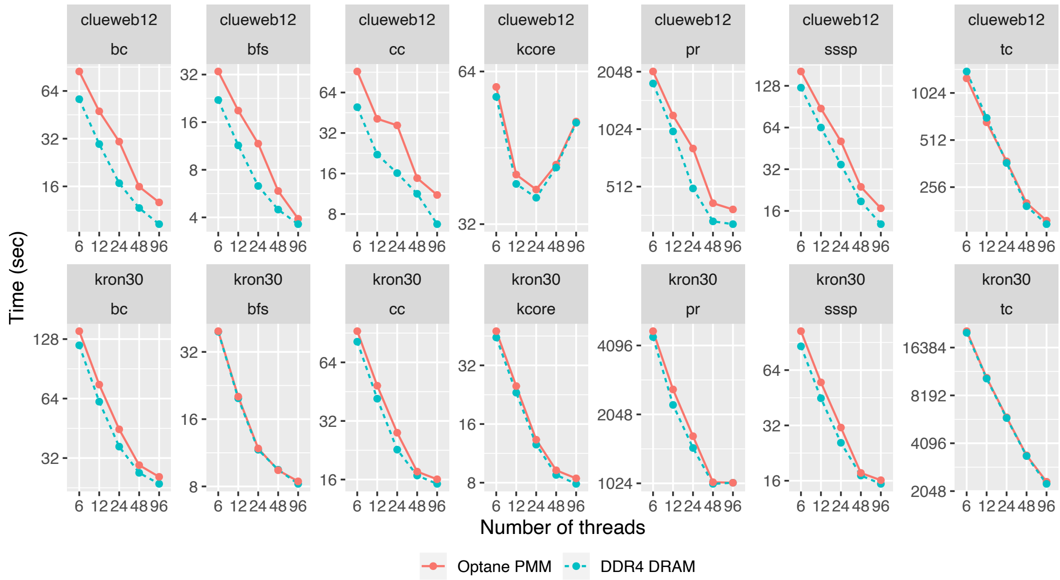


Figure 2.11: Strong scaling in execution time of benchmarks in Galois using DDR4 DRAM and Optane PMM.

GAP [15], GraphIt [190], and GBBS [56] - are evaluated on the Optane PMM machine using several graph analytics applications. Section 2.7.2 describes experiments with medium-sized graphs stored either in Optane PMM or in DRAM. These experiments provide end-to-end estimates of the overhead of executing applications with data in Optane PMM rather than in DRAM. Section 2.7.3 describes experiments with large graphs that fit only in Optane PMM, and performance is compared with distributed-memory execution on a production cluster with up to 128 machines. Section 2.7.4 presents our experiments with GridGraph [196], an out-of-core graph analytics framework, using Optane PMM's app-direct mode to treat it as external memory.

2.7.1 Galois, GAP and GraphIt on Optane PMM

Setup. To choose a shared-memory graph analytics system for our experiments, we evaluate (1) Galois [118], which is a library and runtime for graph processing, (2) GAP [15], which is a benchmark suite of expert-written graph applications, (3) GraphIt [190], which is a domain-specific language (DSL) and optimizing compiler for graph computations, and (4) GBBS [56], which is a benchmark suite of graph algorithms written in the Ligra [151] framework. They exemplify different approaches to shared-memory graph analytics.

GraphIt is a DSL that supports only vertex programs, and it has a sophisticated compiler that uses auto-tuning to generate optimized code; the optimizations are under the control of the programmer. Galois is a C++-based general-purpose programming system based on a runtime that permits optimizations to be specified in the program at compile-time or at runtime, giving the application programmer a large design space of implementations that can be explored. GBBS programs are expressed in a graph processing library and runtime, Ligra. Thus, Galois and GBBS require more programming effort than GraphIt. GBBS includes theoretically efficient algorithms written by experts. GAP is a benchmark suite of graph analytics applications written by expert programmers.

The kcore application is not implemented in GAP and GraphIt, so we omit it in the comparisons reported in this section. We omit the the largest graph wdc12 for GAP and GraphIt because neither of them can handle graphs that have more than $2^{31}-1$ nodes (they use a `signed 32-bit int` for

storing node IDs). GAP, GraphIt, and GBBS do not use NUMA allocation policies within their applications, so we use the OS utility numactl to choose the NUMA interleaved policy. For Galois, we chose the best-performing algorithm using a runtime option, and we did not try different worklists or chunk-sizes. Galois allows application programmers to choose NUMA interleaved or blocked allocation policies for each application by modifying a template argument in the program, and we choose interleaved for bfs, cc, and sssp and blocked for bc, and pr. For GraphIt, we used the optimizations recommended by the authors [190] in the GraphIt artifact.

Results. Figures 2.9 and 2.10 show the execution times of the benchmarks on Optane PMM (GraphIt does not have bc). Galois is generally much faster than GraphIt, GAP, and GBBS: on the average, Galois is 3.8×, 1.9×, and 1.6× faster than GraphIt, GAP, and GBBS respectively. There are many reasons for these performance differences.

Algorithms and implementation choices are part of the story as discussed in Section 2.6.2. For all algorithms, GAP, GBBS and GraphIt use a dense worklist to store the *frontier*, while Galois uses a sparse worklist except for pr (large diameter graphs tend to have sparse frontiers). All systems use the same algorithm for pr. For bfs, all systems, except Galois, use direction-optimization that accesses both in-edges and out-edges (increasing memory accesses). For sssp, GAP, GBBS, and Galois use delta-stepping, while GraphIt does not support such algorithms. For cc, GAP, and GBBS use a union-find based *pointer-jumping* algorithm, Galois uses label-propagation with shortcut-

ting, and GraphIt uses a label propagation algorithm because it supports only vertex programs. Furthermore, Galois uses asynchronous execution for sssp and cc, unlike the others.

Another key difference is the way in which the three systems perform memory allocations. Galois is the only framework that explicitly uses huge pages of size 2MB, whereas GAP, GBBS and GraphIt use small pages of size 4KB and rely on the OS to use *Transparent Huge Pages (THP)*. As discussed in Section 2.5, huge pages can significantly reduce the cost of memory accesses over small pages even when THP is enabled. Galois is also the only one to provide NUMA blocked allocation, and we chose that policy because it performed observably better than the interleaved policy for some benchmarks such as bc and pr (the performance difference was within 18%). In general, we observe that NUMA blocked performs better for topology-driven algorithms, while NUMA interleaved performs better for data-driven algorithms. In addition, GAP, GBBS, and GraphIt allocate memory for both in-edges and out-edges of the graph, while Galois allocates memory only for whichever direction is needed by the algorithm. This not only increases the memory footprint but leads to conflict misses in near-memory when both in-edges and out-edges are accessed.

As Galois generally performs best as shown here, we use it for the rest of our main experiments.

2.7.2 Medium-size graphs: Using Optane PMM vs. DDR4 DRAM

Setup. We used kron30 and clueweb12 (Table 2.3) to measure the end-to-end overhead of using Optane PMM for graphs that are small enough to fit in DRAM (384 GB). We choose the algorithms in Galois that perform best on 96 threads.

Results. Figure 2.11 shows the strong scaling results on DRAM and on Optane PMM with DRAM as cache. kron30 requires $\sim$ 136GB, which is a third of the DRAM available, so Optane PMM delivers performance almost identical to DRAM by caching the graph in DRAM effectively. On the other hand, clueweb12 requires $\sim$ 365GB, which is quite close to the DRAM available, so there are significantly more conflict-misses ($\approx$ 26%) in the near-memory of Optane PMM. On 96 threads, Optane PMM can take up to 65% more execution time than DRAM, but on the average, it takes only 7.3% more time than DRAM.

Another trend is that if the number of threads is less than 24, Optane PMM can be much slower than DRAM because of the way Galois allocates memory. Interleaved and blocked allocation policies in Galois interleave and block among the *threads* and not among the sockets. If the number of threads is less than 24, all threads run in a single socket and all memory ends up being allocated there, leading to under-utilization of the DRAM in the entire system: this results in more conflict-misses in near-memory.

2.7.3 Very large graphs: Using Optane PMM vs. a Cluster

Table 2.4: Execution time (sec) of benchmarks in Galois on Optane PMM (OB) machine using efficient algorithms (non-vertex, asynchronous) and D-Galois on Stampede cluster (DM) using vertex programs with minimum number of hosts that hold the graph (5 hosts for clueweb12, and uk14, and 20 hosts for wdc12). Speedup of Optane PMM over Stampede cluster. Best times are highlighted in green.

Graph	App	Stampede (DM)	Optane PMM (OB)	Speedup (DM/OB)
clueweb12	bc	51.63	12.68	4.07×
	bfs	10.71	6.43	1.67×
	cc	13.70	11.08	1.24×
	kcore	186.03	51.05	3.64×
	pr	155.00	385.64	0.40×
	sssp	33.87	16.58	2.04×
uk14	bc	172.23	11.53	14.9×
	bfs	28.38	7.22	3.93×
	cc	14.56	21.30	0.68×
	kcore	56.08	7.94	7.06×
	pr	82.77	254.95	0.32×
	sssp	52.49	39.99	1.31×
wdc12	bc	775.84	56.48	13.7×
	bfs	71.50	35.25	2.03×
	cc	69.21	76.00	0.91×
	kcore	105.42	49.22	2.14×
	pr	118.01	1706.35	0.07×
	sssp	136.47	118.81	1.15×

Setup. For very large graphs that do not fit in DRAM, the conventional choices are to use either a distributed or an out-of-core system. We focus on distributed execution in this section, using the state-of-the-art D-Galois system [51] on the Stampede2 [155] cluster. To partition graphs between machines, we follow the recommendations of a previous study [65] and use Outgo-

ing Edge Cut (OEC) for 5 and 20 hosts and Cartesian Vertex Cut (CVC) [23] for 256 hosts. On each machine, D-Galois uses the same computation runtime as Galois. D-Galois supports only bulk-synchronous vertex programs with dense worklists, which simplifies communication and synchronization. Therefore, it cannot support some of the more efficient non-vertex programs in Galois. We exclude graph loading, partitioning, and construction time in the reported numbers.

Results. Table 2.4 compares the performance of Optane PMM (on a single machine) running Galois non-vertex, asynchronous programs (referred to as **OB**) with the distributed cluster (Stampede2 [155]) running D-Galois vertex programs and using the minimum number of hosts required to hold the graph in memory (5 hosts for clueweb12, and uk14, and 20 hosts for wdc12; 48 threads per host and referred to as **DM**). We observe that Optane PMM outperforms D-Galois in most of the cases (best times are highlighted), except for pr on clueweb12, uk14, and wdc12 and cc on uk14, and wdc12. Optane PMM gives the geomean speedup of 1.7× over D-Galois, even though D-Galois is using more cores (240 cores for clueweb12, and uk14, and 960 cores of wdc12), and memory bandwidth. In pr, almost all the nodes are updated in every round (similar to the topology driven algorithms), therefore, on the distributed cluster it primarily benefits from the better spatial locality in D-Galois resulting from the partitioning of the graph into smaller local graphs assigned to each host as well as more memory bandwidth.

Further Analysis. For logistical reasons, it is difficult to ensure that both

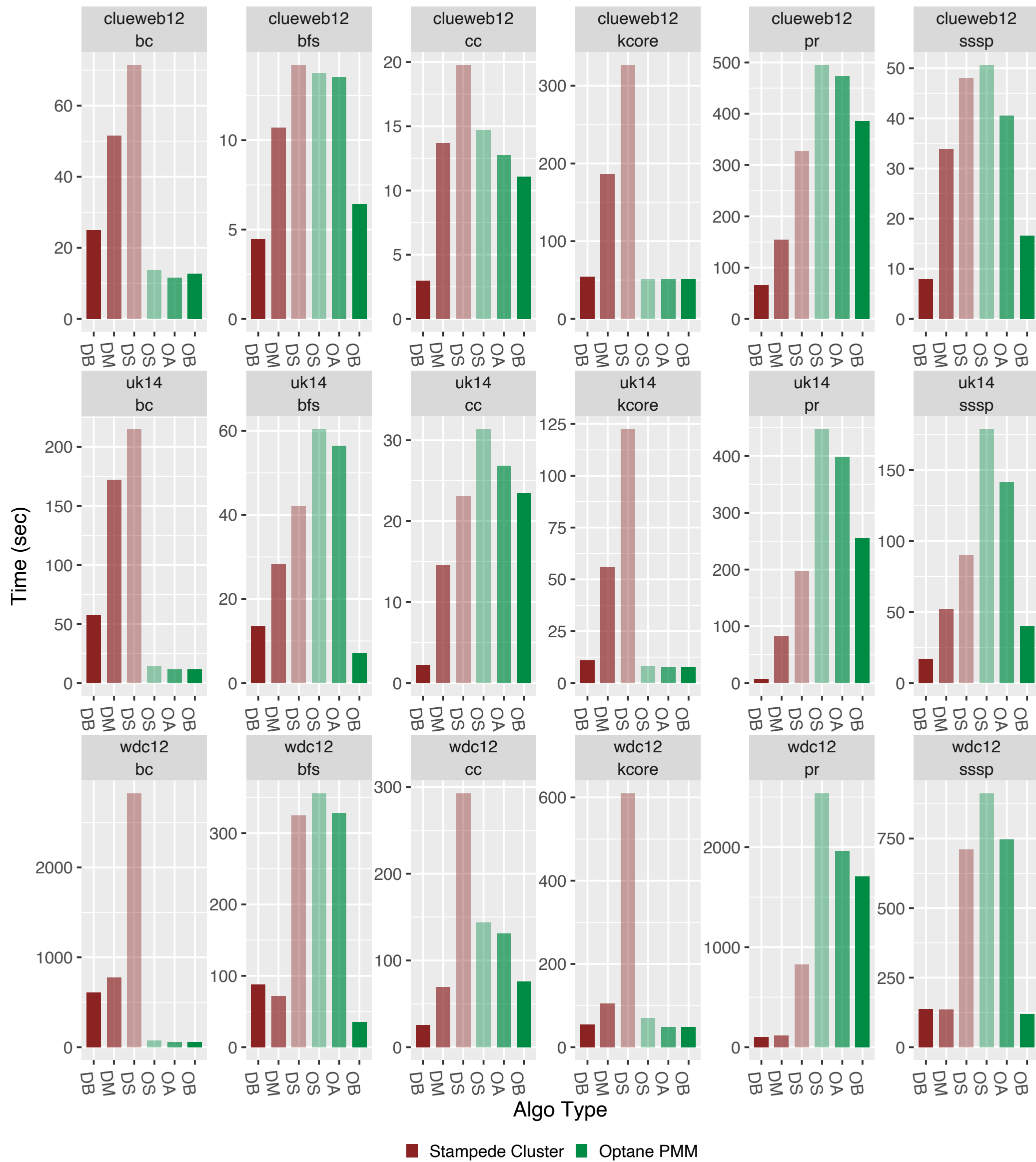


Figure 2.12: Execution time of benchmarks in Galois on Optane PMM machine and D-Galois on Stampede cluster with different configurations :- DB: Distributed Best (all threads on 256 hosts), DM: Distributed Min (all threads on min #hosts that hold graph), DS: Distributed Same (total 80 threads on min #hosts that hold graph), OS: Optane Same (same algorithm and threads as DS), OA: Optane All (same algorithm as DS, DM, and DB on 96 threads), OB: Optane Best (best algorithm on 96 threads).

platforms use the exact same resources (threads and memory). We attempt to get close to a fair comparison by limiting the amount of resources used on both platforms. The bars labeled **O_** in Figure 2.12 show times on the Optane PMM system with the following configurations:- **OB**: Performance using the best algorithm in Galois for that problem and all 96 threads (same as shown in Table 2.4); **OA**: Performance using the best vertex programs in Galois for that problem and all 96 threads; **OS**: Same as **OA** but using only 80 threads. The bars labeled **D_** show times on the Stampede2 system with the following configurations:- **DB**: Performance using D-Galois vertex programs on 256 machines (12,288 threads); **DM**: Performance using D-Galois vertex programs using the minimum number of hosts required to hold graph in memory (same as shown in Table 2.4). **DS**: Same as **DM** but using a total of 80 threads across all machines.

Results. Figure 2.12 shows the results of our experiments. For bars **DS** and **OS**, the algorithm and resources are roughly the same, so in most cases, **OS** is similar or better than **DS**. The only notable exception to this is pr (reason is explained above). On average, **OS** is 1.9× faster than **DS** for all inputs and benchmarks. Bars **OB** and **OA** show the advantages of using non-vertex, asynchronous programs on the Optane PMM system. Bars **DB** and **OB** show that with the more complex algorithms that can be implemented on the Optane PMM system, performance on this system matches the performance of vertex programs on a cluster with vastly more cores and memory for bc, bfs, kcore, and sssp. The main takeaway is that *Optane PMM enables us to perform*

analytics on massive graphs using shared-memory frameworks out-of-the-box while yielding performance comparable or better than that of a cluster with the same resources as the framework may support more efficient algorithms.

2.7.4 Out-of-core GridGraph in App-direct Mode vs. Galois in Memory Mode

In addition to our main shared-memory experiments, we also used an out-of-core graph analytics system with app-direct mode on Optane PMM to examine its performance.

Setup. We used GridGraph [196], a state-of-the-art out-of-core graph analytics framework to compare Optane PMM's app-direct (AD) with Memory Mode (MM) running shared-memory Galois (used throughout this work). The Optane PMM machine was configured in AD mode as described in Section 2.3. In AD, GridGraph manages all of the available DRAM (memory budget given as 384GB) unlike in MM where DRAM is managed by the OS as another level of cache. The input graphs (preprocessed by GridGraph) are stored on the Optane PMM modules which are then used by GridGraph during execution. We used a 512 by 512 grid as the partitioning grid for GridGraph (the GridGraph paper used larger grid partitions for larger graphs to better fit blocks into cache). [3] GridGraph uses a `signed 32-bit int` for storing the node IDs, making it impractical for large graphs with $> 2^{31} - 1$ nodes such as wdc12. We conduct a run of bfs and cc: it does not have bc, kcore, or sssp, and we

[3]We have tried a higher number of partitions, but preprocessing fails as GridGraph opens more file descriptors than the machine supports.

have observed pr failing to complete due to assertion errors in the code.

Table 2.5: Execution time (sec) of benchmarks in Galois on Optane PMM in Memory Mode (MM) and the state-of-the-art out-of-core graph analytics framework GridGraph on Optane PMM in App-direct Mode (AD). A 512 by 512 partition grid was used for GridGraph. Best times are highlighted in green. "—" indicates that system failed to finish in 2 hours.

Graph	App	GridGraph (AD)	Galois (MM)	Speedup (AD/MM)
clueweb12	bfs	5722.75	6.43	890.0×
	cc	5411.23	11.08	488.4×
uk14	bfs	—	7.22	NA
	cc	5700.48	21.30	267.6×

Results. Table 2.5 compares the performance of Optane PMM in Memory Mode (MM) running shared-memory Galois and app-direct mode (AD) running out-of-core GridGraph. We observe that Galois using MM is orders of magnitude faster than GridGraph using AD for bfs, and cc on clueweb12, and cc on uk14 (GridGraph bfs on uk14 failed to finish in 2 hours). This can be attributed to the more sophisticated algorithms (in particular, non-vertex programs and asynchronous data-driven algorithms which are supported in Galois using MM unlike out-of-core frameworks such as GridGraph that only support vertex-programs) and the additional overhead of IO required by out-of-core frameworks, especially for real-world web-crawls with very high diameter such as clueweb12 (diameter $\approx$ 500). We note that after few rounds of computation on bfs for clueweb12, very few nodes get updated: however, the blocks containing those nodes/corresponding edges are still fetched from the storage

to be processed.

2.7.5 Discussion and Summary

While our study was specific to Optane PMM, the guidelines in summarized below apply to other large-memory analytics systems as well.

- Studies using synthetic power-law graphs like kron and rmat can be misleading because unlike these graphs, large real-world web-crawls have large diameters (Section 2.6).
- For good performance on large diameter graphs, the programming model must allow application developers to write work-efficient algorithms that need not be vertex programs and the system must provide data structures for sparse worklists to enable asynchronous data-driven algorithms to be implemented easily (Section 2.6).
- On large-memory NUMA systems, the runtime must manage memory allocation instead of delegating it to the OS. It must exploit huge pages and NUMA blocked allocation. NUMA migration is not useful. (Section 2.5)

2.8 Related Work

Shared-Memory Graph Processing. Shared-memory graph processing frameworks like Galois [118], Ligra [56, 151], Polymer [188] and GraphIt [190] provide users with abstractions to program graph computations that efficiently

leverage a machine's underlying properties such as NUMA, memory locality, and multicores. Shared-memory machines are limited by the amount of available main memory on the system in which it loads the graph into memory for processing: if a graph cannot fit, then out-of-core or distributed processing must be used. However, if the graph fits in memory, the cost of shared memory systems is less than out-of-core or distributed systems as they do not suffer disk reading overhead or communication overhead, respectively.

Optane PMM increases the amount of available memory to shared-memory graph processing systems, and our evaluation shows that algorithms run with Optane PMM are competitive or better than D-Galois [51], a state-of-the-art distributed graph analytics system. This is consistent with past work in which it was shown that shared-memory graph processing on large graphs can be efficient [56], and our findings extend to cases where a user has large amounts of main memory (it is not limited to Optane PMM).

Out-of-core Graph Processing. Out-of-core graph processing systems such as GraphChi [98], X-Stream [141], GridGraph [196], Mosaic [105], Lumos [169], CLIP [10], and BigSparse [87] do graph computation by loading appropriate portions of a graph into memory and writing it back out to disk in a disciplined manner to reduce disk access overhead. Therefore, these systems are not limited by main memory like shared-memory systems. The overhead of disk operations, however, greatly impacts performance of these systems compared to shared-memory systems. The advent of Optane PMM provides an opportunity to use shared-memory systems out-of-the-box as users are able to

increase the main memory to run graph computations on large graphs without a significant performance decrease as our evaluation shows.

Distributed Graph Processing. Distributed graph processing systems such as PowerGraph [70], Gemini [195], D-Galois [51], and others [33, 39, 68, 79, 106, 181] are able to process large graphs by distributing the graph among many machines which increases both available memory as well as computational power. However, since computation is spread among many machines, communication among the machines is required, and this can add significant overhead to the runtime of an algorithm. Additionally, getting access to a distributed cluster can be expensive to an average user. Using Optane PMM to increase the memory available to shared memory systems solves both the memory issue and the cost issue that makes distributed systems difficult to use.

Persistent Memory. Prior work on non-volatile memory includes file systems designed for persistent memory [40, 46, 58, 182], making sure access to persistent memory is efficient while being semantically consistent [45, 94, 109, 168], database systems in persistent memory [12, 166, 187]. Non-volatile memory is expected to improve performance in other areas too.

Optane PMM Evaluation. Since the official release of Optane PMM , many studies have evaluated the potential of Optane PMM for different application domains. Izraelevitz et al. [83] presented a detailed analysis of the performance characteristics of Optane PMM with evaluation on SPEC 2017 benchmarks, various file systems and databases. Optane PMM has also been evaluated for HPC applications with high memory and I/O bottlenecks [174, 178]. Malicevic

et al. [107] use app-direct-like mode to study graph analytics applications using emulated NVM system. However, they only study vertex programs on very small graphs. Peng et al. [126] evaluates the performance of graph analytics applications on Optane PMM in memory mode. However, only use artificial Kronecker [102] and RMAT [36] generated graphs, which, as we have shown in this work, exhibit different structural properties as compared to real-world graphs.

Our study evaluates graph applications on large real-world graphs using more efficient and sophisticated algorithms (in particular, non-vertex programs and asynchronous data-driven algorithms) and is also the first work, to the best of our knowledge, to compare the performance of graph analytics applications on Optane PMM using Galois with the state-of-the-art distributed graph analytics framework, D-Galois [51], on a production level cluster (Stampede [155]) as well as to the out-of-core framework, GridGraph [196], on Optane PMM.

Chapter 3

Abelian Compiler[1]

3.1 Motivation

Graph analytics systems must handle very large data-sets with billions of nodes and trillions of edges [101]. Graphs of this size are too big to fit into the memory of a single machine, so one approach is to use distributed-memory clusters consisting of multicore processors. Writing efficient distributed-memory programs can be difficult, so a number of frameworks and libraries such as Pregel [106], PowerGraph [70], and Gemini [195], have been developed to ease the burden of writing graph analytics applications for such machines. New trends in processor architecture have made this programming problem much more difficult. To reduce energy consumption, computer manufacturers are turning to *heterogeneous* processor architectures in which each machine has a multicore processor and GPUs or FPGAs. To exploit such platforms, we must tackle the twin challenges of *processor heterogeneity* and *distributed-memory computing*. Frameworks like Lux [86] and Gluon [51] permit graph analytics applications writers to use distributed GPUs, but they require writing

[1]This work was originally published in Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Italy, 2018, Proceedings [63]. The first author conceived the key compiler design ideas and abstractions while co-authors helped with its implementation and presentation.

platform-specific programs that are not portable.

Ideally, we would have a compiler that takes single-source, high-level specifications of graph analytics algorithms and automatically translates them into distributed, heterogeneous implementations while optimizing them for diverse processor architectures. This chapter describes such a compiler, called *Abelian*. Application programs are generalized vertex programs written in the Galois programming model, which provides programming patterns and data structures to support graph applications [119]. Section 3.2 describes this programming model in more detail. The Abelian compiler, described in Section 3.3, targets the Gluon runtime [51], which implements bulk-synchronous execution. Unlike other systems in this space, this runtime supports a number of graph partitioning policies including edge-cuts and vertex-cuts, and the programmer can choose any of these policies. The compiler exploits domain-knowledge to generate distributed code, inserting optimized communication code. Back-end compilers generate optimized code for NUMA multi-cores and GPUs from the output of Abelian.

3.2 Programming Model

Abelian supports the programming model described in Chapter 1 Section 1.1.1 which is more general than other systems in this space. In particular, an operator is allowed to update the labels of both the active node *and* its immediate neighbors, which is useful for applications like matrix completion using stochastic gradient descent. In addition, Abelian does not require updates to

node labels to be reduction operations. For example, k-core decomposition evaluated in Section 3.4 uses subtraction on node labels.

In addition to the operator, the programmer must specify how active nodes are found in the graph [116]. The simplest approach is to execute the program in rounds and apply the operator to every node in each round. The order in which nodes are visited is unspecified, and the implementation is free to choose whatever order is convenient. These *topology-driven algorithms* [130] terminate when a global quiescence condition is reached. The Bellman-Ford algorithm for single-source shortest-path (sssp) is an example.

An alternative strategy is to track active nodes in the graph and apply the operator only to those nodes, which potentially creates new active nodes. These *data-driven algorithms* [130] terminate when there are no more active nodes in the graph. As before, the order in which active nodes are to be processed is left unspecified, and the implementation is free to choose whatever order is convenient. Chaotic relaxation sssp uses this style of execution. Tracking of active nodes can be implemented by maintaining a *work-list* of active nodes. Alternatively, this can be implemented by marking active nodes in the graph and making sweeps over the graph, applying the operator only to marked nodes; we call this approach *filtering*. Fine-grain synchronization in marking and unmarking nodes can be avoided by using Jacobi-style iteration with two flags, say *current* and *next*, on each node; in a round, active nodes whose *current* flag is set are processed, and if a node becomes active in that round, its *next* flag is set using an ordinary write operation. The roles

of these flags are exchanged at the end of each round. In our programming model, data-driven algorithms are written using work-lists, but the compiler transforms the code to use a filtering implementation. The correctness of this transformation is ensured by the fact that active nodes can be processed in any order.

3.2.1 Implementation:

This programming model is implemented in C++ using the Galois library [119]. Figure 3.1 shows a program for push-style data-driven algorithm of pagerank. A work-list is used to track active nodes. The **Galois::for_each** in line 30 populates the work-list initially with all nodes in the graph and then iterates over it until the work-list is empty. The operator computes the update to the pagerank of the active node, and it pushes this update to all neighbors of the active node. If the residual at a neighbor exceeds some user-specified threshold, that neighbor becomes active and is pushed to the work-list.

The semantics of the **Galois::for_each** iterator permit work-list elements to be processed in any order. In a parallel implementation of the iterator, each operator application must appear to have been executed atomically. To ensure this, the application programmer must use data structures provided in the Galois library which include graphs, work-lists, and accumulators. This permits the runtime to manage updates to distributed data structures on heterogeneous devices and allows the compiler to treat data structures as objects with known semantics, which enables program optimization and generation of

```
1 struct NodeData{
2   // data on each node
3   unsigned int nout; // out-degree
4   float rank;
5   std::atomic<float> res; // residual
6 };
7
8 struct PageRank {
9   Graph* g;
10  PageRank(Graph* g) : g(g) {}
11  void operator()(GNode src,
12                  Worklist& wl) {
13    auto& sd = g->getData(src);
14    auto res_old=sd.res.exchange(0);
15    // apply residual to self
16    sd.rank += res_old;
17    auto delta=res_old*alpha/sd.nout;
18    for (auto e : g->getEdges(src)) {
19      GNode dst = g->getEdgeDst(e);
20      auto& dd = g->getData(dst);
21      // update residual of dest
22      dd.res += delta;
23      if (dd.res > tolerance) {
24        wl.push(dst);
25      }
26    }
27  }
28 };
29
30 Galois::for_each(g, PageRank{g});
```

Figure 3.1: Pagerank source program

```
1 struct Add_contrib {
2  typedef float ValTy;
3  static ValTy extract(NodeData& node){
4   return node.contrib;
5  }
6  static bool reduce(NodeData& node,
7                     ValTy y) {
8   add(node.contrib, y);
9   return true;
10 }
11 static void reset(NodeData& node) {
12  node.contrib = 0;
13 }
14 };
15
16 struct Bcast_contrib {
17  typedef float ValTy;
18  static ValTy extract(NodeData& node){
19   return node.contrib;
20  }
21  static void setVal(NodeData& node,
22                     ValTy y) {
23   node.contrib = y;
24  }
25 };
```

Figure 3.2: Compiler-generated synchronization structures for field contrib in pagerank

parallel code from implicitly parallel programs as described in Section 3.3.

3.2.2 Restrictions on operators

Like in other programming models for graph analytics [70, 86, 135, 195] and compilers for data-parallel languages [13, 140, 164], operators cannot perform I/O operations. They also cannot perform explicit dynamic memory allocation since some devices (like GPUs) have limited support for this in their runtimes. The library data structures can perform dynamic storage allocation, but this is done transparently to the programmer.

3.3 Abelian Compiler

Figure 3.4 is an overview of how input programs are compiled for execution on distributed, heterogeneous architectures. The Abelian compiler (implemented as a source-to-source translation tool based on Clang's libTooling) analyzes the patterns of data accesses in operators, restructures programs for execution on distributed-memory architectures, and inserts code for optimized communication. The output of the Abelian compiler is a bulk-synchronous parallel C++ program with calls to the Gluon [51] communication runtime (Figure 3.3). Gluon transparently handles the graph partitioning while loading the input graph. The generated code is independent of the partitioning policy, but the partitioning policy determines which portions of this code are executed. This permits Gluon's optimization that exploits structural invariants in partitioning without recompiling the program. The Abelian compiler

```
1 struct NodeData {
2   // data on each node
3   unsigned int nout; // out-degree
4   float rank;
5   float res; // residual
6   // compiler added field
7   std::atomic<float> contrib;
8 };
9 DistributedAccumulator work_done;
10 ... // field-specific bitvector, flags
11 ... // field-specific sync structures
12 struct PageRank {
13   Graph* g;
14   const float &l_alpha, &l_tolerance;
15   ... // copy constructor for members
16   void operator()(GNode src) {
17    auto& sd = g->getData(src);
18    if(sd.res > l_tolerance) {
19      work_done += 1; // do not
      terminate
20      auto res_old = sd.res;
21      sd.res = 0;
22      sd.rank += res_old;
23      Bitvec_rank.set(src);
24      auto delta=res_old*l_alpha/sd.
      nout;
25      for (auto e:g->getEdges(src)) {
26       GNode dst = g->getEdgeDst(e);
27       auto& dd = g->getData(dst);
28       dd.contrib += delta;
29       Bitvec_contrib.set(dst);
30      } } }
31 };
32 struct PageRank_splitOp {
33   Graph* g;
34   PageRank_splitOp(Graph* g) : g(g) {}
35   void operator()(GNode src) {
36     auto& sd = g->getData(src);
37     sd.res += sd.contrib;
38     Bitvec_res.set(src);
39     sd.contrib = 0;
40   }
41 };
42 ... // 1st round for all nodes in
      initial work-list
43 do { // subsequent rounds: predicate-
      based filter
44   work_done.reset(); // for
      termination
45
46   ... // sync res if required: readSrc
47   Galois::do_all(g.getSources(),
48     PageRank{&g,alpha,tolerance});
49   Flag_rank.set_writeSrc();
50   Flag_contrib.set_reduceDst();
51
52   if (Flag_contrib.is_reduceDst()) {
53     graph.sync<reduceDst, readSrc,
54       Add_contrib, Bcast_contrib>
55       (Bitvec_contrib); // executed
56     Flag_contrib.reset_reduceDst();
57   }else if(Flag_contrib.is_reduceSrc()
     ){
58     // sync contrib: reduceSrc,
     readSrc
59   } else{...}// sync contrib if
     required
60   Galois::do_all(g.getSources(),
61                    PageRank_splitOp{&g})
     ;
62   Flag_res.set_writeSrc();
63 } while(work_done.reduce());
```

Figure 3.3: Compiler-generated pagerank program

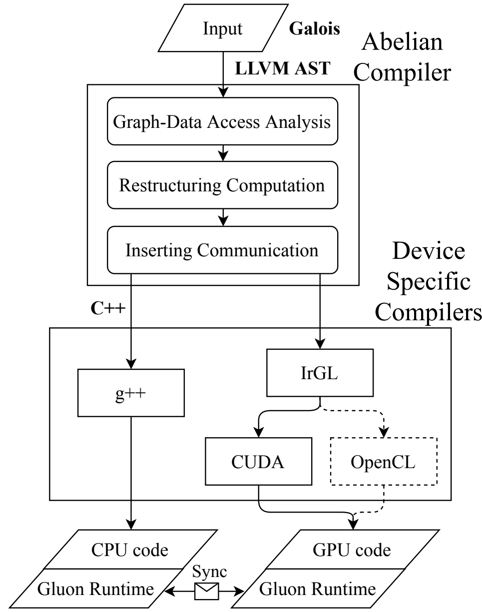


Figure 3.4: System Overview

also generates IrGL [124] intermediate representation kernels corresponding to each Galois::do_all call in the C++ program and inserts code in the C++ program to switch between calling the Galois::do_all and the corresponding IrGL kernel depending on the configuration chosen for the host (these are not shown in Figure 3.3 for brevity). The C++ program and the IrGL intermediate code are then compiled using device-specific compilers. The output executable is parameterized by the graph input, the partitioning policy, and the number of hosts and their configuration (CPU or GPU). The user can thus experiment with a variety of partitioning strategies and heterogeneous devices with a single command-line switch.

3.3.1 Graph-data Access Analysis

The access analysis pass analyzes the fields accessed in an operator. The results of this analysis are used to insert required communication code.

Field accesses are classified as follows:

- *Reduction:* The field is read and updated using a reduction operation inside an edge iterator within the operator (e.g., addition to *residual* in line 22 in Figure 3.1). This is a common and important pattern in graph analytics applications.
- *Read:* The field is read, and it is not part of a reduction (e.g., read from *nout* in line 17 in Figure 3.1).
- *Write:* The field is written, and it is not part of a reduction (e.g., write to *rank* in line 16, Figure 3.1).

In addition, it is useful to abstract the context in which a field access is made.

- *At source:* The field is accessed at the source node of an edge.
- *At destination:* The field is accessed at the destination node of an edge.
- *At any:* The field is accessed at a node independent of any edge or at both endpoints of an edge.

3.3.2 Restructuring computation

The goal of computation restructuring is to bridge the semantic gap between the programming model, which has a single address space, and the execution model, which is distributed-memory and bulk-synchronous parallel. The semantics of Galois iterators permit iterations to be executed in parallel as long as each iteration appears to execute atomically. This fine-grain,

iteration-level parallelism must be converted to round-based, bulk-synchronous parallelism by the Abelian compiler. This includes eliminating global variables (similar to *closure conversion* in functional languages) by adding them as members of the structure. This also requires two key transformations.

3.3.2.1 Splitting operators

When active nodes are processed in parallel on a shared-memory machine, fine-grain synchronization may be needed for correct execution. This problem appears in a different guise on distributed-memory machines: if the two active nodes are on different hosts, proxies will be created on both hosts for the common neighbor, and it is necessary to reconcile the values pushed to these proxies so that the semantics of the program are respected. The bulk-synchronous execution model does not permit fine-grain synchronization, so these kinds of problems must be solved, in general, by breaking up the operator into phases if necessary and introducing `sync` calls between phases. There are a number of cases to consider depending on the type of field access as determined by the graph-data access analysis. We describe this for one such case.

In the PageRank source code in Figure 3.1, the *residual* field is read (line 14) to update the *rank* field (line 16) and written (line 14 using *exchange(0)*) at the *source*, but it is also reduced (line 22) at the *destination*. Since different hosts could update the residual, the hosts reading it should have the reduced value. To handle this, the compiler splits any operator that has such a depen-

dence into multiple operators (a form of *loop fission*): one with only Read and Write accesses to the field and another with only Reduction accesses, as shown in the `PageRank` and `PageRank_splitOp` operators (lines 12-41) respectively in Figure 3.3. This may involve introducing new fields to store the intermediate values (e.g., `contrib`). The compiler also transforms some non-reduction read-after-write operations (e.g., subtraction) to equivalent reduction operations (e.g., addition) in a similar way. After this transformation, `sync` calls are introduced between the parallel phases, as described in Section 3.3.3.

3.3.2.2 Eliminating work-lists

The Abelian compiler eliminates work-lists by using *filtering*, as explained in Section 3.2: in a given round, all nodes in the graph are visited and the operator is applied to nodes whose *current* flag is set. This flag is reset, and if a node becomes active in that round, its *next* flag is set; the roles of the flags are exchanged at the end of each round.

In some algorithms, the predicate used in the source program to push an active node to the work-list can be used during filtering to check if the node is active. Extracting this predicate involves a form of *loop fission*, and it avoids introducing flags and synchronizing their accesses. For example, in Figure 3.1, the code in lines 23-24 adds active nodes to the work-list. In the generated code, this is eliminated, and a new operator is created to conditionally activate nodes as shown in line 18 in Figure 3.3. Another operator is created to execute all nodes that would have been on the initial work-list (line 42). Abelian

can also directly take filter-based implementation of data-driven algorithms as an input, in which case this transformation is not required. Termination is detected using a distributed accumulator (lines 19 and 63) provided by Gluon.

3.3.3 Inserting communication

The final pass of the Abelian compiler inserts code for communication and synchronization. A simple approach is the following: in each round, every mirror sends its value to its master where these values are combined, and the result is broadcast back to all the mirrors. This is essentially the gather-apply-scatter model used by most systems in this space, and it can be implemented by inserting a Gluon [51] `sync` call after each operator for every field that might be updated by that operator. Compilers for heterogeneous systems, such as Falcon [164], Dandelion [140], LiquidMetal [13], and DMLL [32], take a similar approach since their granularity of synchronization is an object or field. This coarse-grained approach can be seen as a more elaborate version of the write-broadcast cache coherence protocol used in systems with hardware cache-coherence. Abelian implements a different, fine-grained communication protocol to reduce the communication volume: a host sends the value of a field to other hosts only if that field has been updated in the previous rounds and if this value will be read in the current round. Static analysis is not adequate to determine these properties, so instrumentation code is inserted to track this dynamically. The actual communication is performed by the Gluon runtime, and it is invoked by inserting sync calls into the code.

3.3.3.1 Fine-grained communication

In graph analytics applications, each round typically updates the field of only a small subset of graph nodes. A device-local, field-specific bit-vector is used to track updates to nodes' fields that participate in communication. The analysis pass determines points in the operator where these fields might be updated, and the compiler inserts instrumentation code at those points to also update the node's bit in the bit-vector for that field (lines 23, 29, 38 in Figure 3.3). The Gluon sync interface permits this bit-vector to be passed to the runtime system, which uses it to avoid sending node values that have not been updated in the current round.

3.3.3.2 On-demand communication

Using the bit-vector ensures only updated values are communicated, but it does not permit Gluon's communication optimization that exploits structural invariants in partitioning policies [51]. To do so, the domain-specific knowledge of abstract write and read locations for the last reduction access(es) and next read access of the field must be specified, respectively. If it is unspecified or imprecise, Gluon may conservatively perform some redundant synchronization. The Abelian compiler can only precisely identify the abstract locations of fields accessed within an operator and cannot be precise about the future accesses. Therefore, after an operator, it inserts code that sets or invalidates the sync-state invalidation flags for fields that could be written in the operator using its write location (lines 49, 50, 62 in Figure 3.3). Before

an operator, it inserts the synchronization structures, as shown in Figure 3.2 (equivalent GPU functions generated for a vector of nodes are omitted for brevity), and the communication code for fields that could be read in the operator (lines 46, 52-59 in Figure 3.3). The code checks the field-specific sync-state flags and calls the Gluon sync routine with the precise write and read locations if the flag is invalidated.

3.3.4 Device-specific compilers

The Abelian compiler outputs C++ code that can be compiled using existing compilers like g++ to execute on shared-memory NUMA multicores using the Galois runtime [119]. A naive translation of this C++ code to CUDA or OpenCL is not likely to yield high-performance code because it will not exploit SIMD execution. We instead use the IrGL [124] compiler, which produces highly optimized CUDA and OpenCL code from an intermediate representation that is intended for graph applications. This compiler exploits nested parallelism, which is important when processing scale-free graphs. To interface with the IrGL compiler, the Abelian compiler generates IrGL intermediate code, translating data layout of fields from arrays of structures to structures of arrays.

3.4 Experimental Evaluation

To evaluate the performance of programs generated by the Abelian compiler, we studied a number of graph analytical applications: betweenness

Table 3.1: Inputs and their key properties

	clueweb12 [134]	**kron30** [102]	**rmat28** [36]	**amazon** [74]
$\|V\|$	978M	1073M	268M	31M
$\|E\|$	42,574M	10,791M	4,295M	82.5M
$\|E\|/\|V\|$	44	16	16	2.7
max D_{out}	7,447	3.2M	4M	44557
max D_{in}	75M	3.2M	0.3M	25366

centrality (bc), breadth-first search (bfs), connected components (cc), k-core decomposition (kcore), pagerank (pr), single-source shortest path (sssp), and matrix completion using stochastic gradient descent (sgd). We specify the programs in Galois C++: pull-style topology-driven algorithm for pr, push-and-pull-style topology-driven algorithm for sgd, and push-style work-list-driven algorithms for the rest. The Abelian compiler analyzes the program, restructures the operators, and synthesizes precise communication. Unless otherwise noted, all optimizations are applied in our evaluation, including eliminating work-lists. The programs work with different partitioning policies. In our evaluation, we choose incoming edge-cut for pr, cartesian vertex-cut for sgd, and outgoing edge-cut for all other benchmarks. We have empirically found these policies to work well in practice; an exhaustive search to find the best policy is outside the scope of this work.

Table 3.1 shows the input graphs we used along with their properties. All the CPU experiments were done on the Texas Advanced Computing Center's [4] Stampede [155] KNL Cluster. For GPU experiments, the Bridges [162] supercomputer at the Pittsburgh Supercomputing Center [3, 161] was used. Table 3.2 shows the configuration of these clusters used in our experiments. In all our experiments, we choose the max-degree node as the source for bc,

bfs, and sssp. For kcore, we solve for $k = 100$. We present the mean execution time of 3 runs, excluding graph partitioning time. *We run pr and sgd for 100 and 50 iterations, respectively; all other algorithms are run until convergence.*

Table 3.2: Cluster configurations

	Stampede (CPU)	**Bridges (GPU)**
NIC	Omni-path	Omni-path
Machine	Intel Xeon Phi KNL	4 NVIDIA Tesla K80s
No. of hosts	32	16
Each host	272 threads	1 Tesla K80
Memory	96GB DDR4	128GB DDR5
Compiler	g++ 7.1	g++ 5.3

Table 3.3: Bridges: execution time (in seconds) on 16 GPUs for rmat28

	D-IrGL	**Abelian**
bc	9.6	9.6
bfs	1.1	1.2
cc	2.6	2.7
kcore	1.5	1.5
pr	32.9	30.5
sssp	2.5	2.5

3.4.1 Comparison with the state-of-the-art

We compare the performance of Abelian compiler-generated programs with handwritten D-Galois programs for CPU-only systems [51] and handwritten D-IrGL programs for GPU-only systems [51]. D-Galois and D-IrGL programs have explicit synchronization specified by the programmer; in contrast, synchronization in programs produced by the Abelian compiler is introduced automatically by the compiler. However, all these programs use Gluon [51],

Table 3.4: Stampede: execution time (in seconds) (H: hosts)

		Gemini		D-Galois		Abelian	
		8H	**32H**	**8H**	**32H**	**8H**	**32H**
bc	**clueweb12**	-	-	OOM	430.4	OOM	437.6
	kron30	-	-	41.3	27.0	39.7	27.3
bfs	**clueweb12**	OOM	69.9	11.6	9.1	12.0	10.1
	kron30	5.1	7.1	5.1	4.0	5.2	4.2
cc	**clueweb12**	39.3	38.8	OOM	16.5	OOM	18.3
	kron30	15.8	14.8	7.6	4.6	7.7	4.0
kcore	**clueweb12**	-	-	OOM	290.4	OOM	289.1
	kron30	-	-	4.4	3.0	4.5	3.0
pr	**clueweb12**	OOM	257.9	395.1	248.0	402.1	277.4
	kron30	245.1	232.4	278.1	221.9	281.0	232.5
sssp	**clueweb12**	OOM	128.3	OOM	14.3	OOM	15.8
	kron30	14.0	14.9	9.4	8.2	9.3	8.2
sgd	**amazon**	-	-	1570.2	701.6	1570.2	696.2

a communication substrate that optimizes communication at runtime by exploiting structural and temporal invariants in partitioning (Gluon uses LCI [50] for message transport between hosts). In addition, D-Galois and Abelian use the same Galois [119] computation operators on the CPU while D-IrGL and Abelian use the same IrGL [124] computation kernels on the GPU. Therefore, differences in performance between Abelian-generated code and D-Galois/D-IrGL code arise mainly from differences in how synchronization code is inserted by the Abelian compiler.

We also compare Abelian-generated programs with distri-buted-CPU programs written in the Gemini framework [195] (Gemini does not have kcore and sgd; bc in Gemini uses bfs while that in Abelian uses sssp, so it is omitted). Gemini has explicit communication messages in the programming model, and it provides a third-party baseline for our study.

Table 3.3 and Table 3.4 show the distributed-GPU and distri-buted-

CPU results. Abelian programs match the performance of D-Galois and D-IrGL programs; the difference is not more than 12%. Gemini is 15% faster than Abelian for pr with kron30 on 8 hosts. In all other cases, Abelian matches or outperforms Gemini. The geometric mean speedup of Abelian over Gemini on 32 KNL hosts is 2.4×. These results show that Abelian is able to compile a high-level, shared-memory, single address space specification into efficient implementations that either match or beat the state-of-the-art graph analytics platform. Although the Abelian compiler produces code for heterogeneous devices, we do not report numbers for distributed CPU+GPU execution because the 4 GPUs on a node on Bridges outperform the CPU by a significant margin.

3.4.2 Impact of communication optimizations

We analyze the performance impact of the communication optimizations in Abelian (Section 3.3.3) by comparing three levels of communication optimization.

1. *Unoptimized* (UO): the Gluon sync call is inserted for a field after an operator if it could be updated in that operator. The bit-vector as well as the abstract write and read locations are left unspecified, so all elements in the field are synchronized. Existing compilers for heterogeneous systems like Falcon [164], Dandelion [140], and Liquid Metal [13] do similar field-specific, coarse-grained synchronization.

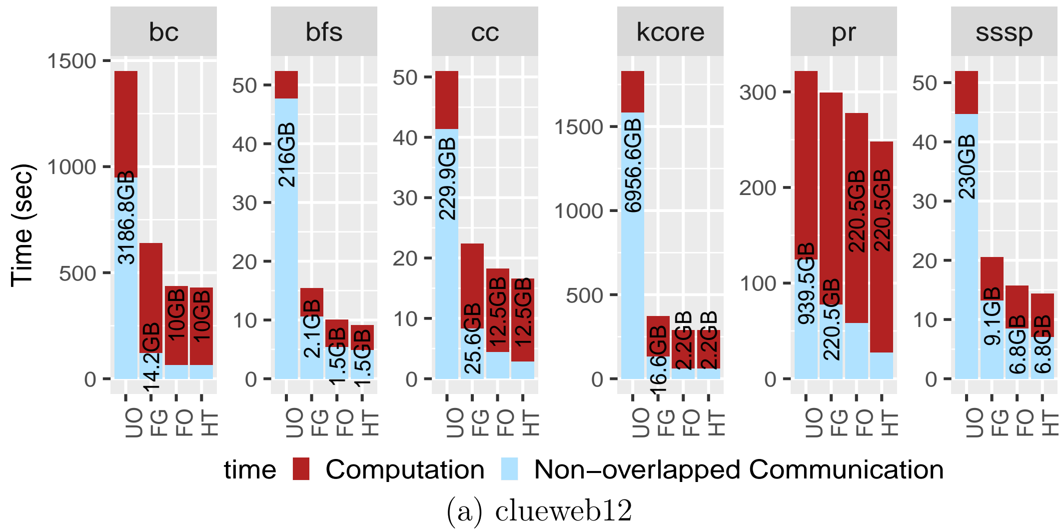
2. *Fine-grained communication optimization* (FG): the compiler instruments

the code to use a bit-vector that dynamically tracks updates to fields. The Gluon sync call used is the same as in UO, but it only synchronizes the elements in the field that have been updated using the bit-vector. This is similar to existing graph analytical frameworks [39, 70, 195] that synchronize only the updated elements.

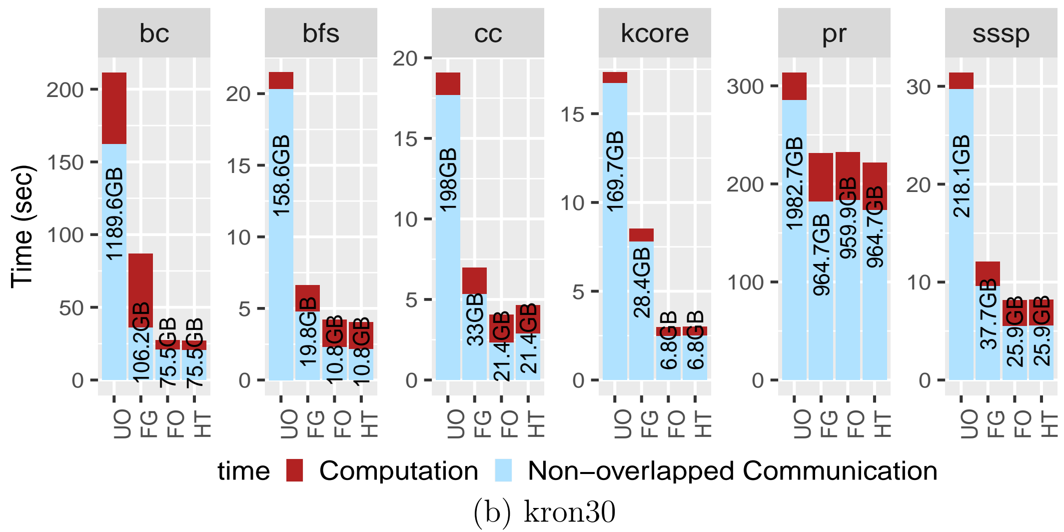
3. *Fine-grained and on-demand communication optimization* (FO): this (default of Abelian compiler) uses on-demand communication along with fine-grained optimization. It instruments invalidation flags to track fields that have been updated and inserts Gluon sync calls before an operator for fields that could be read in the operator, thereby precisely identifying both the abstract write and read locations. This enables Gluon's communication optimization that exploits structural invariants in partitioning policies.

We compare these three communication optimization variants with hand-tuned (HT) programs written in D-Galois and D-IrGL on distributed CPUs and distributed GPUs respectively. In these programs, the programmer (with global control-flow knowledge) specified the precise communication using Gluon sync calls.

Figure 3.5 and Figure 3.6 present the comparison results on 32 KNL hosts of Stampede and 16 GPU devices of Bridges respectively. Each bar in the figures shows the execution time (maximum across hosts). We measure the maximum computation time across hosts in each round and take their sum,



(a) clueweb12



(b) kron30

Figure 3.5: 32 KNL hosts on Stampede: clueweb12 and kron30. Different variants are: UnOpt (UO), Fine-Grained opt (FG), Fine-grained+On-demand opt (FO), Hand-Tuned(HT)

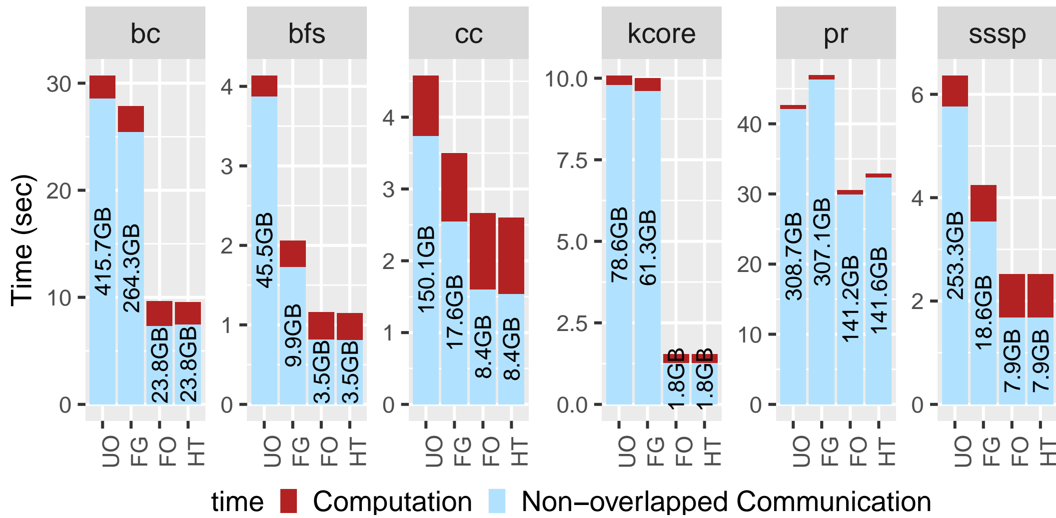


Figure 3.6: 16 GPU devices on Bridges: rmat28

which is the total computation time (top). The rest of the execution time is non-overlapped communication time (bottom). We also measure the total communication volume across all rounds, shown in text on the bars.

The trends are clear in the figure. Each optimization reduces communication volume and time, improving execution time further. FG significantly reduces communication volume and time over UO, with the exception of pr. FG performs atomic updates to the bit-vector, which could be overhead when the updates are dense, like in pr. FO optimizes the communication volume and time further to match the performance of HT. FO reduces communication volume by 23× over UO, yielding a geometric mean execution time speedup of 3.4×. Fine-grained and on-demand communication optimizations (FO) are thus essential to match the performance of HT on both CPUs and GPUs.

Abelian compiler-generated programs can support different partitioning

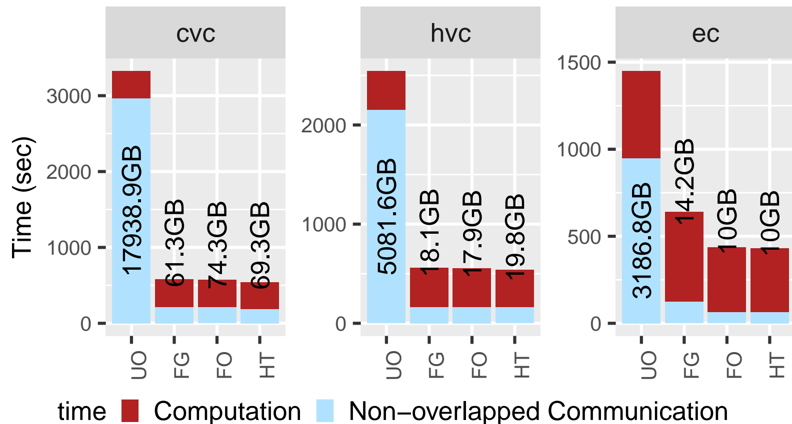


Figure 3.7: 32 KNL hosts on Stampede: partitionings for bc on clueweb12

policies, and we study whether they can fully exploit Gluon's partition-aware optimizations like HT. Figure 3.7 presents the comparison results for bc on clueweb12 using different partitioning policies namely, cartesian vertex cut [23] (cvc), hybrid vertex-cut [39] (hvc), and outgoing edge cut (ec). This shows that FO matches the performance of HT, although FG does not. This shows that the compiler can capture sufficient domain-specific knowledge to aid the Gluon runtime in performing partition-aware optimizations.

3.5 Related Work

Distributed graph processing systems: Many frameworks [39, 51, 70, 86, 106, 181, 195] exist which provide a runtime to simplify writing distributed graph analytics algorithms. Like Abelian, these systems use a vertex programming model and bulk-synchronous parallel (BSP) execution. Abelian is the first compiler that synthesizes the required communication. Our evalu-

ation shows that the programs generated by the Abelian compiler that use the Gluon [51] runtime match hand-tuned programs in the Gluon system and outperform those in the Gemini [195] system.

Single-host heterogeneous graph processing systems: There are several frameworks for graph processing on a single GPU [124], multiple GPUs [16, 125, 194] and multiple GPUs with a CPU [62]. All of these are restricted to a single physical node that connects all devices unlike our system, and consequently, they cannot handle graphs as large as the ones our system can. Abelian leverages the throughput optimizations in the IrGL [124] compiler that are essential for performance on power-law graphs. Unlike IrGL, which compiles an intermediate-level program representation to CUDA, the Abelian compiler not only generates this from a high-level C++ program but also synthesizes synchronization code to execute the compiled code on multiple devices in multiple hosts.

Compilers for distributed or heterogeneous architectures: For heterogeneous architectures, Liquid Metal [13] compiles the Lime language to heterogeneous CPUs, GPUs, and FPGAs. Dandelion [140] compiles high-level LINQ programs to distributed heterogeneous systems. Green-Marl [78] is a DSL that is compiled to Pregel. Brown et al. [32] compile a data-parallel intermediate language DMLL to multicores, clusters, and GPUs. Upadhyay et al. [164] compile a domain-specific language, Falcon, to Giraph code for CPU clusters and MPI+OpenCL code for GPU clusters, but it does not do GPU-specific computation restructurings like nested parallelism which Abelian compiler does

using IrGL. In all these compilers, the granularity of communication is an object or field, whereas Abelian identifies fine-grained elements of a label-array and communicates them precisely using the Gluon runtime. Moreover, none of the existing compilers use domain-specific analysis and computation restructurings for graph analytical applications like Abelian.

Chapter 4

Experimental Study of Partitioning Policies[1]

4.1 Motivation

An important runtime parameter to the compiler-generated distributed code in Chapter 3 is the partitioning policy, which is left for the user to choose. The application performance is sensitive to the partitioning policy chosen. When a graph is partitioned, a node in the graph may be replicated on several machines, and communication is required to keep these replicas synchronized. Good partitioning policies attempt to reduce this synchronization overhead while keeping the computational load balanced across machines. A number of recent studies have looked at ways to control replication of nodes, but these studies are not conclusive because they were performed on small clusters with eight to sixteen machines, did not consider work-efficient data-driven algorithms, or did not optimize communication for the partitioning strategies they studied.

[1]This work was originally published in the proceedings of VLDB Endowment, 12(4), 2018 [66]. The development of the key ideas as well as implementation of all the partitioning policies used in this work were done by the first author while co-authors helped with the experimentation, data collection, and presentation.

4.2 Background

Graph partitioning must balance two concerns:

The first concern is *computational load balance*. For the graph algorithms considered in this work, computation is performed in rounds: in each round, *active* nodes in the graph are visited, and a computation is performed by reading and writing the immediate neighbors of the active node [101]. For simple *topology-driven* graph algorithms in which all nodes are active at the beginning of a round, the computational load of a host is proportional to the numbers of nodes and edges assigned to that host, so by dividing nodes and edges evenly among hosts, it is possible to achieve load balance [39]. However, work-efficient graph algorithms like the ones considered in this work are *data-driven*: nodes become active in data-dependent, statically unpredictable ways, so a statically balanced partition of the graph does not necessarily result in computational load balance.

The second concern is *communication overhead*. We consider graph partitioning with replication: when a graph is partitioned, its edges are divided up among the hosts, and if edge $(n_1 \rightarrow n_2)$ is assigned to a host h, proxy nodes are created for n_1 and n_2 on host h and connected by an edge. A given node in the original graph may have proxies on several hosts in the partitioned graph, so updates to proxies must be synchronized during execution using inter-host communication. This communication entirely dominates the execution time of graph analytics applications on large scale clusters as shown by the results in Section 4.6, so optimizing communication is the key to high performance.

Substantial effort has gone into designing graph partitioning strategies that reduce communication overhead. Overlapping communication with computation (as done in HPC applications) would reduce its relative overhead, but there is relatively little computation in graph analytics applications. Therefore, reducing the volume of communication has been the focus of much effort in this area. Since communication is needed to synchronize proxies, reducing the average number of proxies per node (known in the literature as the *average replication factor*) while also ensuring computational load balance can reduce communication [28, 89, 152, 154, 163]. This is one of the driving principles behind the Vertex-Cut partitioning strategy (Vertex-Cuts and other partitioning strategies are described in detail in Section 4.4) used in PowerGraph [70] and PowerLyra [39]. Partitioning policies developed for efficient distribution of sparse matrix computation such as 2D block partitioning [23, 35] can also be used since sparse graphs are isomorphic to sparse matrices.

Several papers [6, 23, 39, 167, 195] have studied how the performance of graph analytics applications changes when different partitioning policies are used, and they have advocated particular partitioning policies based on their results. We believe these studies are not conclusive for the following reasons.

- Most of the evaluations were done for small graphs on small clusters, so it is not clear whether their conclusions extend to large graphs and large clusters.
- In some cases, only topology-driven algorithms were evaluated, so it is not clear whether their conclusions extend to work-efficient data-driven

algorithms.

- The distributed graph analytics systems used in the studies optimize communication only for particular partitioning strategies, putting other partitioning strategies at a disadvantage.

4.3 Contributions

This study makes the following contributions:

1. We present the first detailed performance analysis of state-of-the-art, work-efficient graph analytics applications using different graph partitioning strategies including Edge-Cuts, 2D block partitioning strategies, and general Vertex-Cuts on large-scale clusters, including one with 256 machines and roughly 69K threads. These experiments use a system called D-Galois, a distributed-memory version of the Galois system [118] based on the Gluon communication runtime [51]. Gluon performs communication optimizations that are specific to each partitioning strategy. Our results show that although Edge-Cuts perform well on small-scale clusters, a 2D partitioning policy called Cartesian Vertex-Cut (CVC) [23] performs the best at scale *even though it results in higher replication factors and higher communication volumes than the other partitioning strategies.*

2. We present an analytical model for estimating communication volumes for different partitioning strategies, and an empirical study using micro-

benchmarks to estimate communication times. These help estimate and understand the performance differences in communication required by the partitioning strategies. In particular, these models explain why at scale, CVC has lower communication overhead even though it performs more communication.

3. We give a simple decision tree that can be used by the user to select a partitioning strategy at runtime given an application and the number of distributed hosts. Although the chosen policy might not be the best in some cases, we show that the application's performance using the chosen policy is almost as good as using the best policy.

4.3.1 Lessons:

This work's contributions include some important lessons for designers of high-performance graph analytics systems.

1. It is desirable to support optimized implementations of *multiple* partitioning policies including Edge-Cuts and Cartesian Vertex-Cuts, like D-Galois does. Existing systems either support general partitioning, using approaches like gather-apply-scatter (PowerGraph, PowerLyra), without optimizing communication for particular partitioning policies like Edge-Cuts, or support only Edge-Cuts (Gemini). Neither approach is flexible enough.

2. An important lesson for designers of efficient graph partitioning poli-

cies is that the communication overhead in graph analytics applications depends on not only the communication volume but also the communication pattern among the partitions as explained in Section 4.5. The replication factor and the number of edges/vertices split between partitions [39, 70, 152, 163] are not adequate proxies for communication overhead.

4.4 Partitioning Policies

The graph partitioning policies considered in this work divide a graph's edges among hosts and creating proxy nodes on each host for the endpoints of the edges assigned to that host. If edge $e : (n_1 \rightarrow n_2)$ is assigned to a host h, h creates proxy nodes on itself for nodes n_1 and n_2, and adds an edge between them. For each node in the graph, one proxy is made the *master*, and the other proxies are made *mirrors*. Figure 4.1a shows an example graph and Figure 4.1b shows one of the many ways of partitioning it among 4 hosts. In Figure 4.1b, colored nodes with solid border are master proxies whereas nodes without color and dotted border are mirror proxies. Intuitively, the master holds the canonical value of the node during the computation, and it communicates that value to the mirrors as needed. Each partitioning strategy represents choices made along two dimensions: (i) how edges are partitioned among hosts and (ii) how the master is chosen from the proxies of a given node. To understand these choices, it is useful to consider both the graph-theoretic (i.e., nodes and edges) and the adjacency matrix representation of a graph.

4.4.1 1D Partitions

In 1D partitioning, nodes are partitioned among hosts. If a host owns node n, all outgoing (or incoming) edges connected to n are assigned to that host, and the corresponding proxy for node n is made the master. In the graph analytical literature, this partitioning strategy is called outgoing (or incoming) Edge-Cut [89, 152, 154, 195]. Figure 4.1c shows an example of OEC for the example graph 4.1a along with the communication pattern (for push-style operators) required for OEC i.e reduce among hosts during synchronization. In matrix-theoretic terms, this corresponds to assigning rows (or columns) to hosts. 1D partitioning is commonly used in stencil codes in computational science applications; the mirror nodes are often referred to as *halo nodes* in that context. Stencil codes use topology-driven algorithms on meshes, which are uniform-degree graphs, and computational load balance can be accomplished by assigning roughly equal numbers of nodes to all hosts. For power-law graphs, the adjacency matrix is irregular, and ensuring load balance for data-driven graph algorithms through static partitioning is difficult.

A number of policies are used in practice.

1. *Balanced nodes*: Assign roughly equal numbers of nodes to all hosts.

2. *Balanced edges*: Assign nodes such that all hosts have roughly the same number of edges.

3. *Balanced nodes and edges* [195]: Assign nodes such that a given linear

combination of the number of nodes and edges on a host has roughly the same value on all hosts.

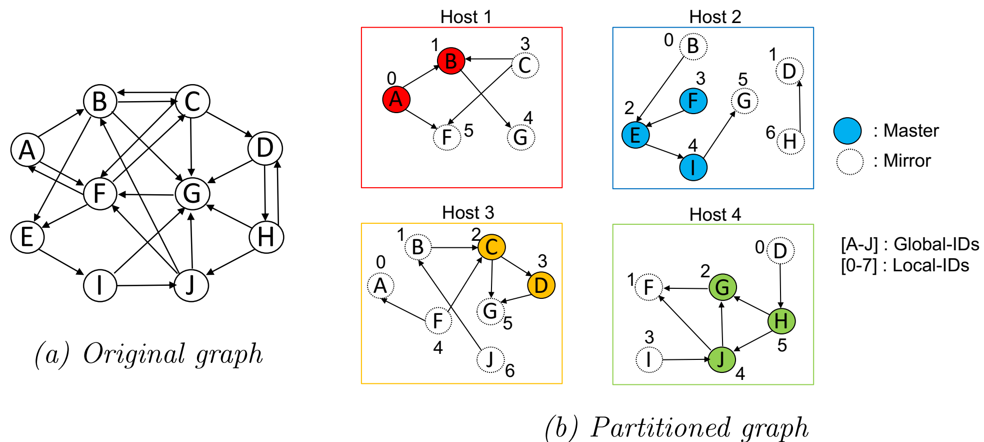
The first policy is the simplest and does not require any analysis of the graph. However, it may result in substantial load imbalance for power-law graphs since there is usually a large variation in node degrees. The other two policies require computing the degree of each vertex and the prefix-sums of these degrees to determine how to partition the set of nodes.

4.4.2 2D Block Partitions

In 2D block partitioning, the adjacency matrix is blocked along both dimensions, and each host is assigned some of the blocks. Unlike in 1D partitioning, both outgoing and incoming edges of a given node may be distributed among different hosts. In the graph analytical literature, such partitioning strategies are called *Vertex-Cuts*. 2D block partitioning can be viewed as a restricted form of Vertex-Cuts in which the adjacency matrix is blocked.

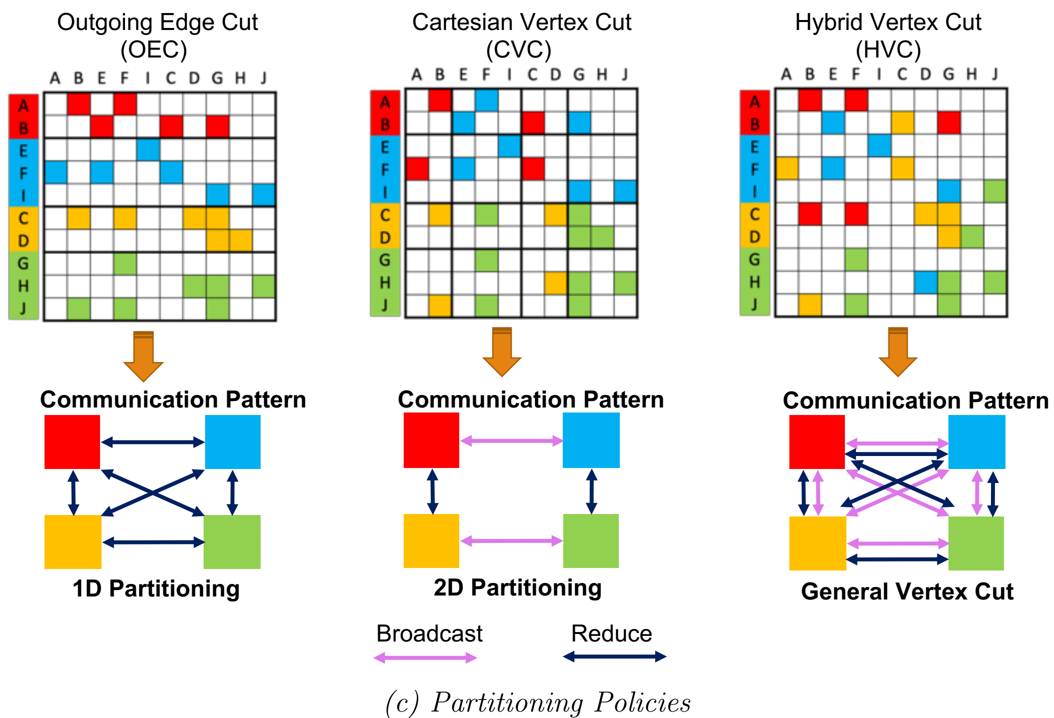
This work explores three alternative implementations of 2D block partitioning, illustrated in Figure 4.2 using a cluster of eight hosts in a 4×2 grid. The descriptions below assume that hosts are organized in a grid of size $p_r \times p_c$.

1. *CheckerBoard Vertex-Cuts (BVC)* [35, 97]: The nodes of the graph are partitioned into equal size blocks and assigned to the hosts. Masters are created on each host for its block of nodes. The matrix is partitioned into contiguous blocks of size $N/p_r \times N/p_c$, and each host is assigned



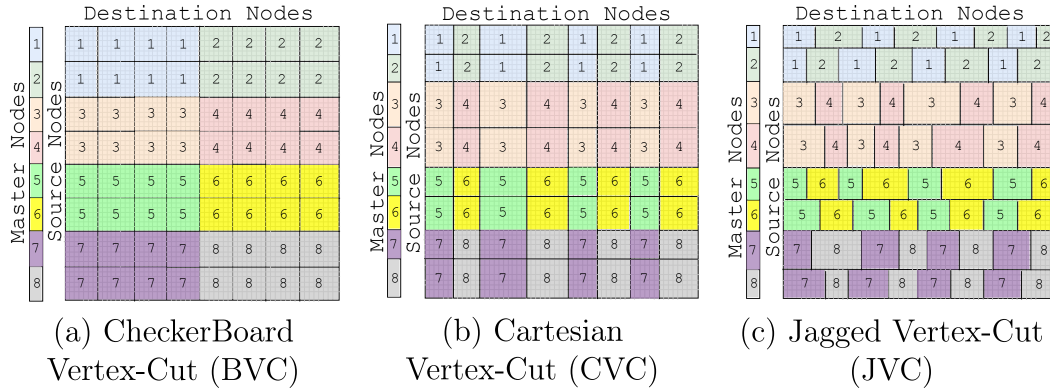
(a) Original graph

(b) Partitioned graph



(c) Partitioning Policies

Figure 4.1: An example of partitioning a graph for 4 hosts (hosts depicted by different colors).



(a) CheckerBoard Vertex-Cut (BVC) (b) Cartesian Vertex-Cut (CVC) (c) Jagged Vertex-Cut (JVC)

Figure 4.2: 2D partitioning policies.

one block of edges as shown in Figure 4.2(a). This approach is used by CombBLAS [33] for sparse matrix-vector multiplication (SpMV) on graphs. There are several variations to this approach; the one used in this study is shown in Figure 4.2(a).

2. *Cartesian Vertex-Cuts (CVC)* [23]: Nodes are partitioned amo-ng hosts using any 1D block partitioning policy, and masters are created on hosts for the nodes assigned to it. Unlike BVC, these node partitions need not be of the same size, as shown in Figure 4.2(b). This can happen, for example, if the 1D block partitioning assigns nodes to hosts such that the number of edges is balanced among hosts.

   The columns are then partitioned into same sized blocks as the rows. Therefore, blocks along the diagonal will be square, but other blocks may be rectangular unlike in BVC. These blocks of edges can be distributed among hosts in different ways. This study uses a block distribution along rows and a cyclic distribution along columns, as shown in Figure 4.2(b).

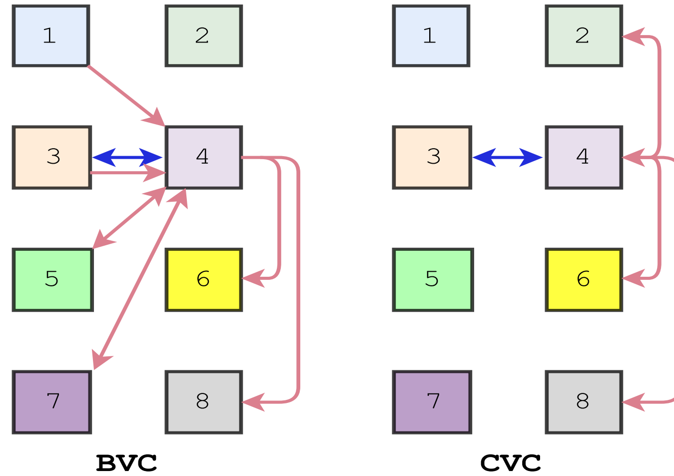


Figure 4.3: Communication patterns in BVC and CVC policies: red arrows are reductions and blue arrows are broadcasts for host 4.

3. *Jagged Vertex-Cuts (JVC)* [35]: The first stage of JVC is similar to CVC. However, instead of partitioning columns into same sized blocks as the rows, each block of rows can be partitioned independently into blocks for a more balanced number of edges (non-zeros) in edge blocks. Therefore, blocks will not be aligned among the column dimension. These edge blocks can be assigned to hosts in any fashion, but in this study, the host assignment is kept the same as CVC as shown in Figure 4.2(c).

Although these 2D block partitioning strategies seem similar, they have different communication requirements.

Consider a push-style graph algorithm in which active nodes perform computation on their own labels and push values to their immediate outgoing neighbors, where they are reduced to compute labels for the next round. In a distributed-memory setting, a proxy with outgoing edges of a node n on host

h push values to its proxy neighbors also present on h. These proxies may be masters or mirrors, and it is useful to distinguish two classes of mirrors for a node n: *in-mirrors*, mirrors that have incoming edges, and *out-mirrors*, mirrors that have outgoing edges. For a push-style algorithm in the distributed setting, the computation at an active node in the original graph is performed partially at each in-mirror, the intermediate values are reduced to the master, and the final value is broadcast to the out-mirrors. Therefore, the communication pattern can be described as broadcast along the row dimension and reduce along the column dimension of the adjacency matrix as shown in Figure 4.1c for CVC. The hosts that participate in broadcast and/or reduce depends on the 2D partitioning policy and the assignment of edge blocks to the hosts.

To illustrate this, consider Figure 4.3 which shows the 4×2 grid of hosts and the reduce and broadcast partners for host 4 in BVC and CVC. For BVC, by walking down the fourth block column of the matrix in Figure 4.2(a), we see that incoming edges to the nodes owned by host 4 may be mapped to hosts $\{1,3,5,7\}$, so labels of mirrors on these hosts must be communicated to host 4 during the reduce phase. Similarly, the labels of mirrors on host 4 must be communicated to masters on hosts $\{5,6,7,8\}$. Once the values have been reduced at masters on host 4, they must be sent to hosts that own out-mirrors for nodes owned by host 4. Walking over the fourth block row of the matrix, we see that only host 3 is involved in this communication. Similarly, the labels of masters on the host $\{3\}$ must be sent to host 4.

The same analysis can be done on CVC and JVC partitionings. For

JVC, a given host may need to involve all other hosts in reductions and broadcasts, leading to larger communication requirements than CVC.

4.4.3 Unrestricted Vertex-Cut Partitions

Unrestricted or *general* Vertex-Cuts are partitioning strategies that assign edges to hosts without restriction, and they do not correspond to 1D or 2D blocked partitions. The partitioning strategies used in the PowerGraph [70] and PowerLyra systems [39] are examples of this strategy. For example, in PowerLyra's *Hybrid Vertex-Cut* (HVC), nodes are assigned to hosts in a manner similar to an Edge-Cut. However, high-degree nodes are treated differently from low-degree nodes to avoid assigning all edges connected to a high-degree node to the same host, which creates load imbalance. If $(n_1 \rightarrow n_2)$ is an edge and n_2 has low in-degree (based on some threshold), the edge is assigned to the host that owns n_2; otherwise, it is assigned to the node that owns n_1. Figure 4.1c shows HVC for graph 4.1a along with its communication pattern which requires reduction and broadcast among all the hosts as it lacks any structural constraints.

4.4.4 Discussion

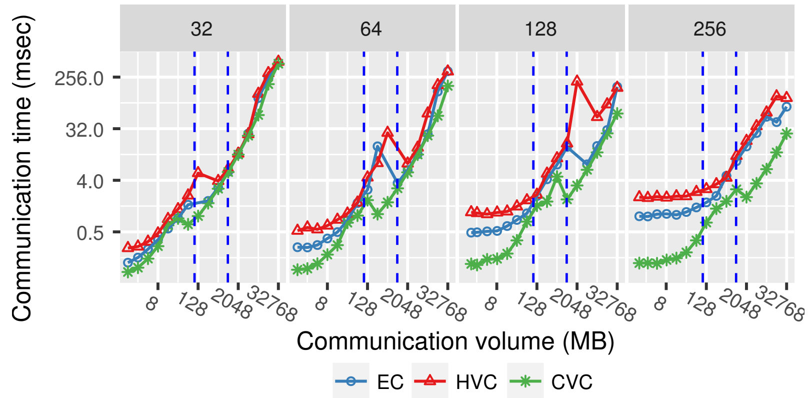
A small detail in 2D block partitioning is that when the number of processors is not a perfect square, we factorize the number into a product $p_x \times p_y$ and assign the larger factor to the row dimension rather than the column dimension. This heuristic reduces the number of broadcast partners because the

reduce phase communicates only updated values from proxies to the master and the number of these updates decrease over iterations, whereas the broadcast phase in each round sends the updated canonical value from the master to all its proxies.

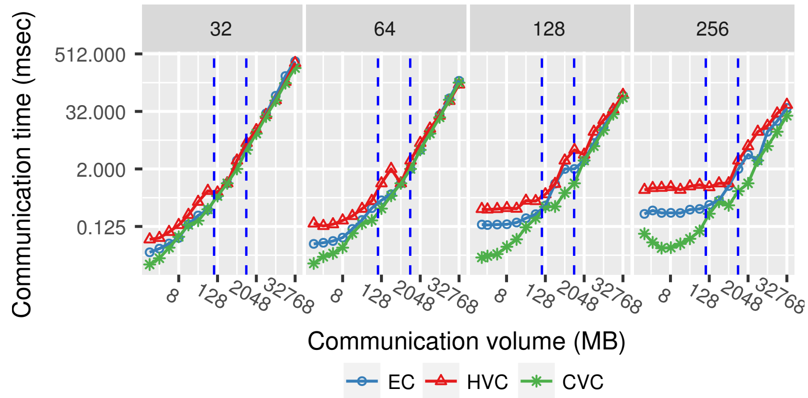
In our studies, we observed that for Edge-Cuts and HVC, almost all pairs of hosts have proxies in common for some number of nodes. Therefore, for these partitioning policies, Edge-Cut partitioning performs all-to-all communication while HVC performs two rounds of all-to-all communication (from mirrors to masters followed by masters to mirrors). On the other hand, CVC has p_c number of parallel all-to-all communications among $p_r \approx \sqrt{P}$ hosts followed by p_r number of parallel all-to-all communications among $p_c \approx \sqrt{P}$ hosts. Therefore, each host in CVC sends fewer messages and has fewer communication partners than EC and HVC, which can help both at the application level (a host need not wait for another host in another row) and at the hardware level (less contention for network resources). The next section explores these differences among partitioning policies quantitatively.

4.5 Performance Model for communication

This section presents performance models to estimate communication volume and communication time for three partitioning policies: Edge-Cut, Hybrid Vertex-Cut, and Cartesian Vertex-Cut. We formally define replication factor (Section 4.5.1), describe a simple analytical model for estimating the communication volume using the replication factor (Section 4.5.2), and use



(a) KNL hosts



(b) Skylake hosts

Figure 4.4: Performance of different communication patterns on different number of hosts on Stampede for different CPU architectures.

micro-benchmarks to estimate the communication time from the communication volume (Section 4.5.3).

4.5.1 Replication Factor

Given a graph $G{=}(V, E)$, and a strategy S for partitioning the graph between P hosts, let v be a node in V and let $r_{P,S}(v)$ denote the number of proxies of v created when the graph is partitioned. $r_{P,S}(v)$ is referred to

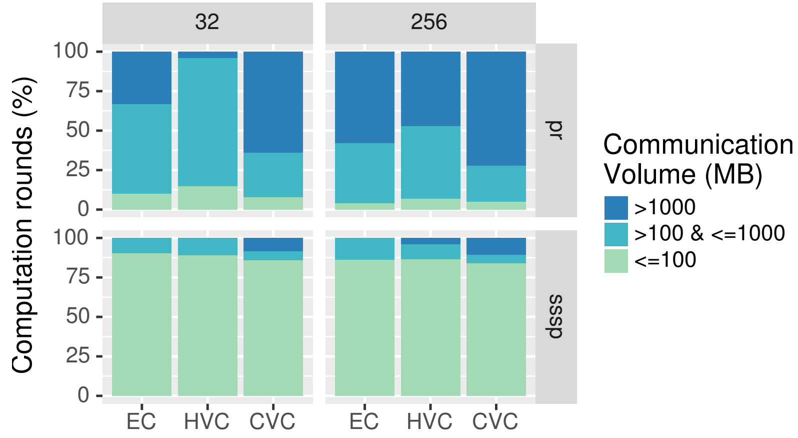


Figure 4.5: Communication volume of each computation round on clueweb12.

as the *replication factor*, and it is an integer between *1* and P. The average replication factor across all nodes is a number between 1 (no node is replicated) and P (all nodes are replicated between all hosts), and it is given by the following equation:

$$\bar{r}_{P,S} = \frac{\sum_{v \in V} r_{P,S}(v)}{|V|} \tag{4.1}$$

4.5.2 Estimating Communication Volume

For simplicity in the communication volume model, assume that the data flow in the algorithm is from the source to the destination of an edge. A similar analysis can be performed for other cases. Assume that u nodes ($0 \leq u \leq |V|$) are updated in a given round and that the size of the node data (update) is b bytes.

4.5.2.1 Edge-Cut (EC)

Consider an incoming Edge-Cut (IEC) strategy (1D column partitioning), and let $r_{P,E}(v)$ be the replication factor for a node v. In IEC, the destination of an edge is always a master, so only the master node is updated during computation. At the end of a round, the master node updates its mirrors on other hosts. Since an update requires b bytes, the master node for v in the graph must send $(r_{P,E}(v) - 1) * b$ bytes. If u nodes are updated in the round, the volume of communication is the following:

$$C_{IEC}(G, P) \approx (\overline{r}_{P,E}-1)*u*b \tag{4.2}$$

For an outgoing Edge-Cut (OEC) strategy (1D row partitioning), the source of an edge is always a master, so the mirrors must send the updates to their master, and only the master needs the updated value. The communication volume remains the same as IEC's only if all the mirrors of u nodes are updated in the round. In practice, only some mirrors of the nodes in u are updated. Let f_E denote the average fraction of mirrors of nodes in u that are updated. The communication volume in the round is the following:

$$C_{OEC}(G, P) \approx (f_E * \overline{r}_{P,E}-1) * u * b \tag{4.3}$$

4.5.2.2 Hybrid Vertex-Cut (HVC)

In a general Vertex-Cut strategy like Hybrid Vertex-Cut, the mirrors must communicate with their master, and the masters must communicate with their mirrors. If $\overline{r}_{P,H}$ is the average replication factor, the volume of

communication in a round in which an average fraction f_H of mirrors are updated is

$$C_{HVC}(G,P) \approx ((f_H * \overline{r}_{P,H} - 1) + (\overline{r}_{P,H} - 1)) * u * b \tag{4.4}$$

Comparing this with (4.2) and (4.3), we see that the volume of communication for OEC and IEC can be much lower than HVC if they have the same replication factor, and it will be greater than HVC only if their replication factor is almost twice that of HVC.

4.5.2.3 Cartesian Vertex-Cut (CVC)

Consider a Cartesian Vertex-Cut with $P = p_r * p_c$. Unlike HVC, at most p_r mirrors must communicate with their master, and the masters must communicate with at most p_c of their mirrors. Let $\overline{r}_{P,C}$ be the average replication factor, and let f_C and f'_C be the fractions of mirrors that are active along rows and columns respectively. Then, the volume of communication in a round is

$$C_{CVC}(G,P) \approx ((f_C * \overline{r}_{P,C} - 1) + (f'_C * \overline{r}_{P,C} - 1)) * u * b \tag{4.5}$$

Comparing this with (4.4), we see that the communication volume for CVC can be less than the volume for HVC even with the same replication factor. This illustrates that the replication factor is not the sole determinant for communication volume.

4.5.3 Micro-benchmarking to Estimate Communication Time

To relate communication volumes under different communication patterns to communication time, we adapted the MVAPICH2 all-to-allv micro-benchmark. For a given total communication volume across all hosts, we simulate the communication patterns of the three strategies (the message sizes differ for different strategies): (1) IEC/OEC: one round of all-to-all communication among all hosts, (2) HVC: two rounds of all-to-all communication among all hosts, (3) CVC: a round of all-to-all communication among all row hosts followed by a round of all-to-all communication among all column hosts. In a given strategy, the same message size is used between every pair of hosts. In practice, the message sizes usually differ and point-to-point communication (instead of a collective) is typically used to dynamically manage buffers and parallelize serialization and deserialization of data.

Figures 4.4(a) and 4.4(b) show the communication time of the different strategies on different number of KNL hosts and Skylake hosts, respectively, on the Stampede cluster[155] (described in Section 4.6) as the total communication volume increases. As expected, the *communication time is dependent not only on the communication volume but also on the communication pattern*. The performance difference in the communication patterns grows as the number of hosts increases. While EC, HVC, and CVC perform similarly on 32 KNL hosts, CVC gives a geomean speedup of 4.6× and 5.6× over EC and HVC respectively to communicate the same volume on 256 KNL hosts; the speedup is higher for low volume ($\leq$ 100MB) than for medium volume ($>$ 100MB and

$\leq$ 1000MB). The communication volume in work-efficient graph analytical applications changes over rounds and several rounds have low communication volume in practice. For example, Figure 4.5 shows the percentage of computation rounds of pagerank and sssp with communication volume in the low, medium, and high ranges for different partitioning policies on 32 and 256 KNL hosts of Stampede. In Figure 4.4, CVC communicates more data in the same amount of time on 256 KNL hosts; e.g., CVC can synchronize 128 to 256 MB of data in the same time that EC and HVC can synchronize 1 MB of data. Similar behavior is also observed on Skylake hosts. Thus, CVC can reduce communication time over EC and HVC for the same communication volume or can increase the volume without an increase in time.

4.5.4 Discussion

The analytical model and micro-benchmark results described in this section show that although communication time depends on the communication volume (which in turn depends on the average replication factor), it is incorrect to conclude from this alone that partitioning strategies with larger average replication factors or communication volume will perform worse than those with smaller replication factors or communication volumes. In particular, CVC might be expected to perform better at scale because it requires fewer messages and fewer pairs of processors to communicate than other partitioning strategies like EC and HVC do. While the micro-benchmarks used MPI collectives with the same message sizes, most distributed graph analytics

Table 4.1: Inputs and their properties.

	kron30	**clueweb12**	**wdc12**
$\|V\|$	1073M	978M	3,563M
$\|E\|$	10,791M	42,574M	128,736M
$\|E\|/\|V\|$	16	44	36
Max D_{out}	3.2M	7,447	55,931
Max D_{in}	3.2M	75M	95M
Size on disk	136GB	325GB	986GB

Table 4.2: Execution time of Gemini and D-Galois with EC.

	32 hosts	**Gemini (sec)**	**D-Galois (sec)**
kron30	**bfs**	7.8	3.0
	cc	16.0	4.8
	pr	213.2	211.6
	sssp	17.5	6.1
clueweb12	**bfs**	72.9	8.9
	cc	38.0	16.9
	pr	231.9	219.6
	sssp	115.8	13.1

systems use MPI or similar interfaces to perform point-to-point communication with variable message sizes. For example, the D-Galois [51] system uses LCI [50][2] instead of MPI for sending and receiving point-to-point messages between hosts. Even in such systems, CVC can be expected to perform better at scale than EC and HVC because it needs fewer messages and fewer pairs of processors to communicate. We study this using the D-Galois system quantitatively in the next section.

4.6 Experimental Evaluation

All experiments were conducted on the Texas Advanced Computing Center's Stampede Cluster (Stampede2) [4, 155]. Stampede2 has 2 distributed clusters: one with Intel Knights Landing (KNL) nodes and another with Intel Skylake nodes. Each KNL cluster host has 68 1.4 GHz cores with 4 hardware threads per core, 96 GB RAM, and 16 GB MC-DRAM, which serves as a direct-mapped L3 cache. Each Skylake cluster host has 48 2.1 GHz cores on 2 sockets (24 cores per socket) with 2 hardware threads per core and 192 GB RAM. Both clusters use 100Gb/sec Intel Omni-Path (OPA) network. We limit our experiments to 256 hosts on both the clusters and we use 272 threads per host (69632 threads in total) on the KNL cluster and 48 threads per host (12288 threads in total) on the Skylake cluster. All code was compiled using g++ 7.1.0. Unless otherwise stated, all results are presented using the KNL cluster.

We used three graphs in our evaluation: synthetically generated randomized power-law graph kron30 (using kron [102] generator with weights of 0.57, 0.19, 0.19, and 0.05, as suggested by graph500 [1]) and the largest publicly available web-crawl graphs, clueweb12 [19, 20, 134] and wdc12 (Web Data Commons) [110, 111]; we present results for wdc12 only at scale (256 hosts). Properties of these graphs are listed in Table 4.1.

To study the impact of graph partitioning on application execution

[2] LCI [50] is an alternative to MPI that has been shown to perform better than MPI for graph analytical applications.

Table 4.3: Different initial Edge-Cut policies used for different benchmarks, inputs, and partitioning policies: IE: Incoming Edge-Cut, OE: Outgoing Edge-Cut, UE: Undirected Edge-Cut.

		bc/bfs/sssp	**cc**	**pr**
kron30	**XEC**	OE	UE	IE
	EC	IE	UE	IE
	HVC	IE	UE	IE
	BVC	OE	UE	IE
	JVC	OE	UE	IE
	CVC	OE	UE	IE
clueweb12	**XEC**	OE	UE	IE
	EC	OE	UE	OE
	HVC	OE	UE	OE
	BVC	OE	UE	IE
	JVC	OE	UE	IE
	CVC	OE	UE	IE
wdc12	**XEC**	OE	UE	IE
	EC	OE	UE	OE
	HVC	OE	UE	OE
	BVC	OE	UE	IE
	JVC	OE	UE	IE
	CVC	OE	UE	IE

time, we use five graph analytics applications: betweenness centrality (bc), breadth-first search (bfs), connected components (cc), pagerank (pr), and single-source shortest path (sssp). We implement topology-driven algorithm for pagerank and data-driven algorithms for the rest. We run bc with only one source. The source nodes for bc, bfs, and sssp are the maximum out-degree node. The directed graph is given as input to bc, bfs, pagerank, and sssp (bc and sssp use a weighted graph), while an undirected graph (we make the directed graph symmetric by adding reverse edges) is given as input to cc. We present the mean execution time of 3 runs (each being maximum across all hosts) excluding graph partitioning time. *All algorithms are run until conver-*

gence except for pagerank, which is run for up to 100 rounds or iterations.

4.6.1 Implementation

Existing graph analytics systems implement a single partitioning strategy, so they are not suitable for comparative studies of graph partitioning. Moreover, these frameworks do not optimize communication patterns to exploit structure in the partitioning policy. Therefore, we used a system called D-Galois: it uses the shared-memory Galois system [101, 118] for computing on each host and the Gluon communication runtime [51] for inter-host communication and synchronization. Gluon is an efficient bulk-synchronous communication substrate, which enables existing shared-memory CPU and GPU graph analytics frameworks to run on distributed heterogeneous clusters. Gluon is partition-aware and optimizes communication for particular partitioning strategies by exploiting their *structural invariants*. Instead of naively reducing from mirrors to masters and broadcasting from masters to mirrors during synchronization (as done in gather-apply-scatter model), it avoids redundant reductions or broadcasts by exploiting structural invariants in the partitioning strategy, as described in Section 4.5. Gluon also exploits the fact that the partitioning of the graph does not change during computation and it uses this *temporal invariance* to reduce the overhead and volume of communication by optimizing metadata like node IDs that needs to be communicated along with the node values or updates.

To ensure that D-Galois is a suitable platform for this study, we com-

pared it with Gemini [195], a state-of-the-art system that uses only Edge-Cuts. Table 4.2 shows the execution times for Gemini versions of our benchmarks on 32 KNL hosts; wdc12 is omitted because Gemini runs out of memory while partitioning it (even on 256 hosts). Since Gemini supports only Edge-Cuts, we compare it with D-Galois using Edge-Cut (EC). We see that D-Galois outperforms Gemini. Gemini also does not scale beyond 32 hosts [51]. These results support the claim that D-Galois is a reasonable state-of-the-art platform for performing studies of graph partitioning.

For our study, graphs are stored on disk in CSR and CSC formats (size for directed, weighted graphs in Table 4.1). The synthetic graphs (kron30) are stored after randomizing the vertices (randomized order) while the web-crawl graphs (clueweb12 and wdc12) are stored in their natural (crawled) vertex order. D-Galois's distributed graph partitioner reads the input graph such that each host only reads a distinct portion of the file on disk once.

D-Galois also supports loading partitions directly from disk, and we use this to evaluate XtraPulp [152], the state-of-the-art graph partitioner for large scale-free graphs. XtraPulp generates (using command line parameters "`-e 1.1 -v 10.0 -c -d -s 0`" and 272 OpenMP threads) partitions for kron30 and clueweb12, and it runs out of memory for wdc12 (the authors have been informed about this). We write these partitions to disk and load it in D-Galois. We term this the XtraPulp Edge-Cut (XEC) policy.

We implement five partitioning policies in D-Galois: (1) Edge-Cut (EC), (2) Hybrid Vertex-Cut (HVC), (3) CheckerBoard Vertex-Cut (BVC),

Table 4.4: Graph partitioning time (includes time to load and construct graph) and static load balance of edges assigned to hosts on 256 KNL hosts.

		Partitioning time (sec)			**Max-by-mean edges**		
		bc bfs sssp	**cc**	**pr**	**bc bfs sssp**	**cc**	**pr**
kron30	**XEC**	304	448	312	1.05	1.08	1.06
	EC	51	76	51	1.01	1.01	1.01
	HVC	102	130	101	1.02	1.02	1.02
	BVC	345	379	365	1.02	1.02	1.02
	JVC	1006	1006	1016	1.00	1.00	1.00
	CVC	261	288	241	1.00	1.00	1.00
clueweb12	**XEC**	381	647	373	3.18	8.93	14.69
	EC	27	152	38	1.00	1.11	1.00
	HVC	308	374	308	3.39	1.64	3.39
	BVC	1179	12907	12843	20.24	20.24	20.24
	JVC	1904	1924	1960	1.82	1.53	1.01
	CVC	573	1239	1119	9.16	2.03	3.26
wdc12	**XEC**	OOM	OOM	OOM	OOM	OOM	OOM
	EC	109	251	236	1.00	1.03	1.00
	HVC	3080	2952	3068	1.18	1.13	1.18
	BVC	8039	OOM	OOM	15.44	OOM	OOM
	JVC	5263	6570	8890	1.09	1.05	1.01
	CVC	2487	4276	3221	1.79	1.17	1.27

(4) Jagged Vertex-Cut (JVC), and (5) Cartesian Vertex-Cut (CVC). Direction of edges traversed during computation is application dependent. bc, bfs, and sssp traverse outgoing edges of a directed graph, pr traverses incoming edges of a directed graph, and cc traverses outgoing edges of a symmetric, directed graph (Gluon handles undirected graphs in this way). Due to this, each application might prefer a different EC policy:

- Outgoing Edge-Cut (OE): Nodes of a directed graph are partitioned into contiguous blocks while trying to balance outgoing edges and all

outgoing edges of a node are assigned to the same block as the node, like in Gemini [195].

- Incoming Edge-Cut (IE): Nodes of a directed graph are partitioned into contiguous blocks while balancing incoming edges and all incoming edges are assigned to the same block.
- Undirected Edge-Cut (UE): Nodes of a symmetric, directed graph are partitioned into contiguous blocks while trying to balance outgoing edges and all outgoing edges are assigned to the same block.

EC does not communicate during graph partitioning as each process reads its portion of the graph directly from the file. *All Vertex-Cut policies use some EC*[3] to further partition the edges of some vertices, which are sent to the respective hosts. Table 4.3 shows the initial Edge-Cut used by the partitioning policies for each benchmark and input. HVC as described in Section 4.4.3 is used for graphs with skewed in-degree (e.g., wdc12 and clueweb12). For graphs with skewed out-degree (e.g., kron30), if $(n_1 \rightarrow n_2)$ is an edge and n_1 has low out-degree (based on some threshold), the edge is assigned to the host that owns n_1; otherwise, it is assigned to the node that owns n_2. BVC, JVC, and CVC are as described in Section 4.4.2.

[3]The Vertex-Cut policies could also have used XEC instead of EC to further partition the edges of some vertices, but we chose EC to show that CVC can do well at scale even with a simple Edge-Cut.

Table 4.5: Dynamic load balance: maximum-by-mean computation time on 256 KNL hosts.

		bc	bfs	cc	pr	sssp
kron30	**XEC**	3.4	1.47	3.49	1.75	1.66
	EC	1.06	1.64	3.60	1.71	1.63
	HVC	1.09	1.70	1.37	1.61	1.57
	BVC	1.16	1.54	1.55	1.19	1.48
	JVC	1.17	1.50	1.55	1.17	1.43
	CVC	1.19	1.45	1.50	1.16	1.36
clueweb12	**XEC**	2.96	2.09	11.81	27.80	2.14
	EC	33.19	2.17	28.76	4.12	2.08
	HVC	8.86	2.12	3.03	7.93	2.26
	BVC	5.04	2.39	19.50	32.46	4.09
	JVC	21.17	2.13	5.69	3.23	2.08
	CVC	7.71	2.20	5.42	6.73	2.65
wdc12	**XEC**	OOM	OOM	OOM	OOM	OOM
	EC	—	2.20	6.31	13.30	2.19
	HVC	—	2.06	2.31	6.32	2.05
	BVC	OOM	1.94	OOM	OOM	OOM
	JVC	—	1.59	1.78	1.81	1.59
	CVC	—	1.67	2.74	6.53	1.75

4.6.2 Partitioning Time

Although the time to partition graphs is an overhead in distributed-memory graph analytics systems, shared-memory systems must also take time to load the graph from disk and construct it in memory. For example, Galois [118] and other shared-memory systems like Ligra [151] take roughly 2 minutes to load and construct rmat28 (35GB), which is relatively small compared to the graphs used in this study. This time should be used as a point of reference when considering the graph partitioning times shown in Table 4.4 for different partitioning policies and inputs on 256 KNL hosts. XEC excludes time to write partitions from XtraPulp to disk, load it in D-Galois, and con-

struct the graph. All other policies include graph loading and construction time. Note that some partitioning policies run out-of-memory (OOM). EC is a lower bound for partitioning as each host loads its partition of the graph directly from disk in parallel. Vertex-Cut policies are slower than Edge-Cut policies because they involve communication and more analysis on top of EC.

Comparing partitioning time for different partitioning strategies is not the focus of this work. In particular, graphs can be partitioned once, and different applications can run using these partitions. The main takeaway for partitioning time is that it can finish within a few minutes. CVC partitioning time is better than or similar to that of XEC. CVC takes around 5 minutes, 10 minutes, and 40 minutes for directed kron30, clueweb12, and wdc12, respectively.

Table 4.6: Execution time (sec) for different partitioning policies, benchmarks, and inputs on KNL hosts.

		32 hosts					
		XEC	**EC**	**HVC**	**BVC**	**JVC**	**CVC**
kron30	**bc**	32.7	40.9	51.7	75.5	26.0	**22.6**
	bfs	4.6	3.0	4.4	5.1	**2.6**	2.9
	cc	10.1	4.8	10.7	13.9	5.6	**4.2**
	pr	230.1	211.6	293.0	287.8	**191.9**	339.9
	sssp	9.7	6.1	8.3	10.4	5.6	**4.8**
clueweb12	**bc**	**136.2**	439.1	627.9	961.9	660.6	539.1
	bfs	10.4	**8.9**	17.4	41.1	38.7	19.2
	cc	OOM	16.9	**7.5**	OOM	25.9	19.6
	pr	272.6	219.6	**193.5**	OOM	354.6	217.9
	sssp	16.5	**13.1**	26.6	63.7	63.2	31.7
		256 hosts					
		XEC	**EC**	**HVC**	**BVC**	**JVC**	**CVC**
kron30	**bc**	23.6	39.1	51.2	20.4	13.6	**10.0**
	bfs	3.8	2.4	4.0	1.3	1.3	**1.0**
	cc	4.3	4.2	3.9	2.4	2.2	**1.9**
	pr	**57.6**	64.2	84.1	65.7	78.7	72.9
	sssp	5.4	3.8	5.9	2.0	2.0	**1.8**
clueweb12	**bc**	413.7	420.0	1012.1	404.6	627.6	**266.9**
	bfs	36.4	27.1	46.5	25.1	36.1	**14.6**
	cc	43.0	84.7	8.4	21.2	12.5	**7.3**
	pr	286.7	82.3	97.5	267.4	**58.6**	60.8
	sssp	43.0	32.5	54.7	44.5	44.7	**21.8**

4.6.3 Load Balance

We first compare the quality of the resulting partitions. The number of edges assigned to a host represents the static load of that host. Table 4.4 shows the maximum to mean ratio of edges assigned to hosts. We see that

Table 4.7: Execution statistics of wdc12 on 256 KNL hosts.

		EC	HVC	JVC	CVC
Execution Time (sec)	**bfs**	422.8	832.9	974.6	**373.4**
	cc	118.8	135.8	178.4	**74.2**
	pr	230.9	193.1	173.3	**138.3**
	sssp	633.1	1238.3	1395.6	**567.9**
Replication Factor	**bfs**	1.4	2	4.6	3.4
	cc	4.4	2.3	7.2	5.3
	pr	1.4	2.0	5	3.1
	sssp	1.4	2	4.6	3.4
Total Communication Volume (GB)	**bfs**	15	27	101	54
	cc	100	36	278	147
	pr	604	628	3019	1394
	sssp	66	159	697	352

static load balance not only varies with different policies but also with the input graphs. For both directed and undirected graphs, kron30 is very well balanced for all policies. For clueweb12 and wdc12, EC is well balanced while the rest are not.

Dynamic load may be quite different from static load, especially for data-driven algorithms in which computation changes over rounds. To measure the dynamic load balance, we measure the computation time of each round on each host and calculate the maximum and mean across hosts. Summing up over rounds, we get the maximum and mean computation time respectively. Table 4.5 shows the maximum-by-mean computation time for all policies, benchmarks, and inputs on 256 hosts. Note that bc, bfs, and sssp use the same partitions. Firstly, it is clear that although kron30 is statically well balanced for all policies, it is not dynamically load balanced. Moreover, the load balance depends on the dynamic nature of the algorithm. Even though bc

Table 4.8: Execution time (sec) on Skylake hosts for EC and CVC.

		8 hosts		256 hosts	
		EC	**CVC**	**EC**	**CVC**
kron30	**bc**	45.9	**35.9**	21.1	**5.5**
	bfs	**3.3**	3.4	4.0	**0.7**
	cc	**8.2**	9.4	7.1	**1.0**
	pr	**230.3**	259.2	38.1	**28.4**
	sssp	**6.3**	7.1	6.0	**1.2**
clueweb12	**bc**	**339.1**	669.2	197.7	**152.4**
	bfs	**6.0**	16.1	6.1	**6.0**
	cc	10.9	**9.5**	64.9	**3.6**
	pr	**121.5**	130.1	20.6	**17.9**
	sssp	**16.9**	33.2	**8.6**	9.7

and bfs use the same graph, their dynamic load balance is different. CVC on clueweb12 is well-balanced for bfs, but highly imbalanced for bc. The policy significantly impacts dynamic load balance, but this need not directly correlate with their static load balance. For example, for bfs and sssp, although CVC on clueweb12 is statically severely imbalanced while EC is not, both EC and CVC are fairly well balanced at runtime. This demonstrates that dynamic load balance is difficult to achieve since it depends on the interplay between policy, algorithm, and graph.

4.6.4 Execution Time

Table 4.6 shows the execution time of D-Galois with all policies on 32 and 256 KNL hosts for kron30 and clueweb12. Table 4.7 shows the execution time for wdc12 on 256 KNL hosts (all policies run out of memory on 32 hosts; XEC and BVC run out of memory on 256 hosts too). These tables also high-

light which policy performs best for a given application, input, and number of hosts (scale). It is clear that the best performing policy is dependent on the application, the input, and the number of hosts (scale). Although there is no clear winner for all cases, CVC performs the best on 256 hosts for almost all applications and inputs.

Table 4.8 shows the execution time of D-Galois with EC and CVC on 8 and 256 Skylake hosts for kron30 and clueweb12. Even on Skylake hosts, the best performing policy depends on the application, the input, and the number of hosts (scale), without any clear winner. Nonetheless, similar to KNL hosts, CVC performs the best on 256 hosts for almost all applications and inputs, whereas EC performs the best on 8 hosts for almost all applications and inputs. This suggests that the relative merits of the partitioning policies are not specific to the CPU architecture.

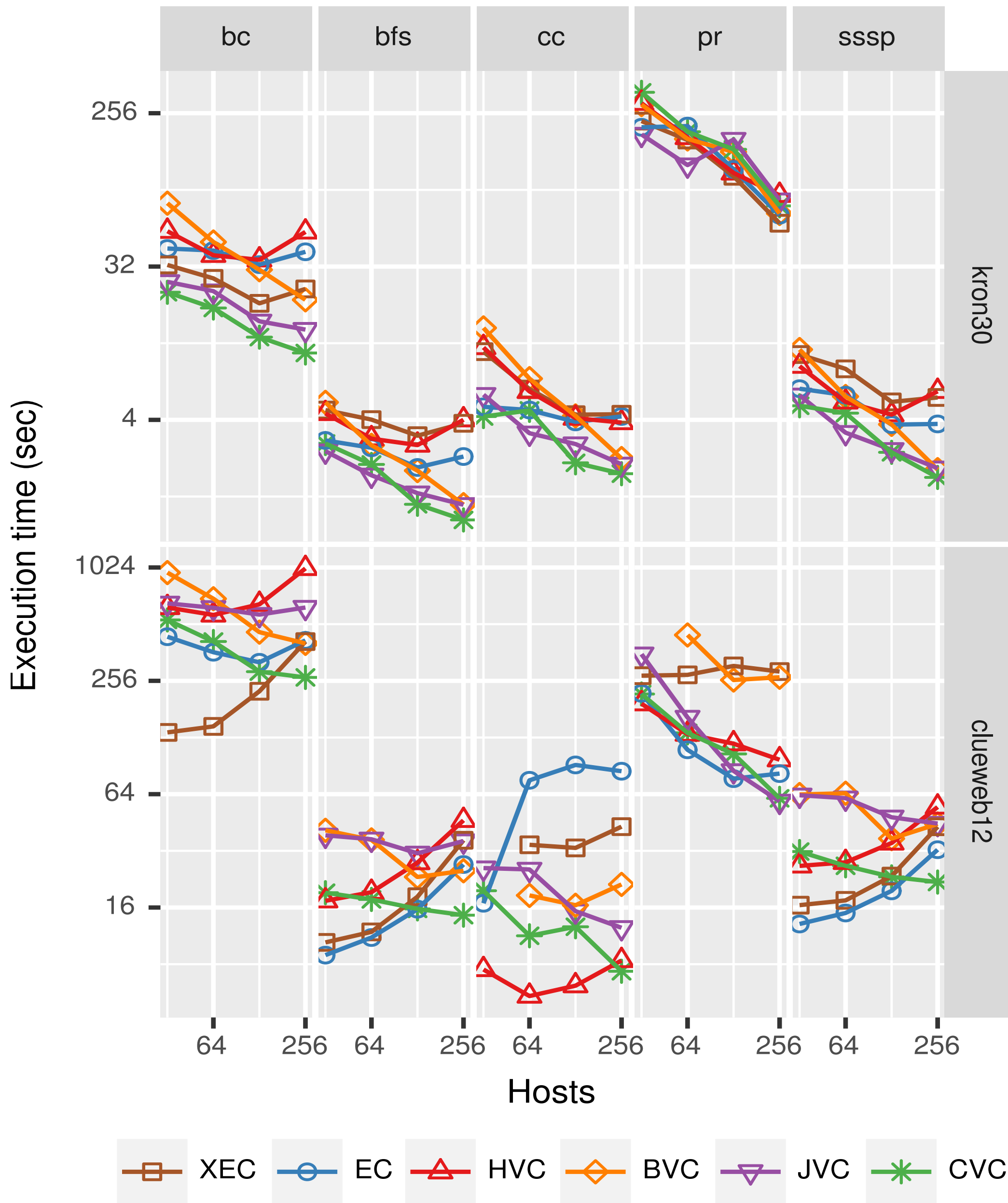


Figure 4.6: Execution time of D-Galois with different partitioning policies on KNL hosts.

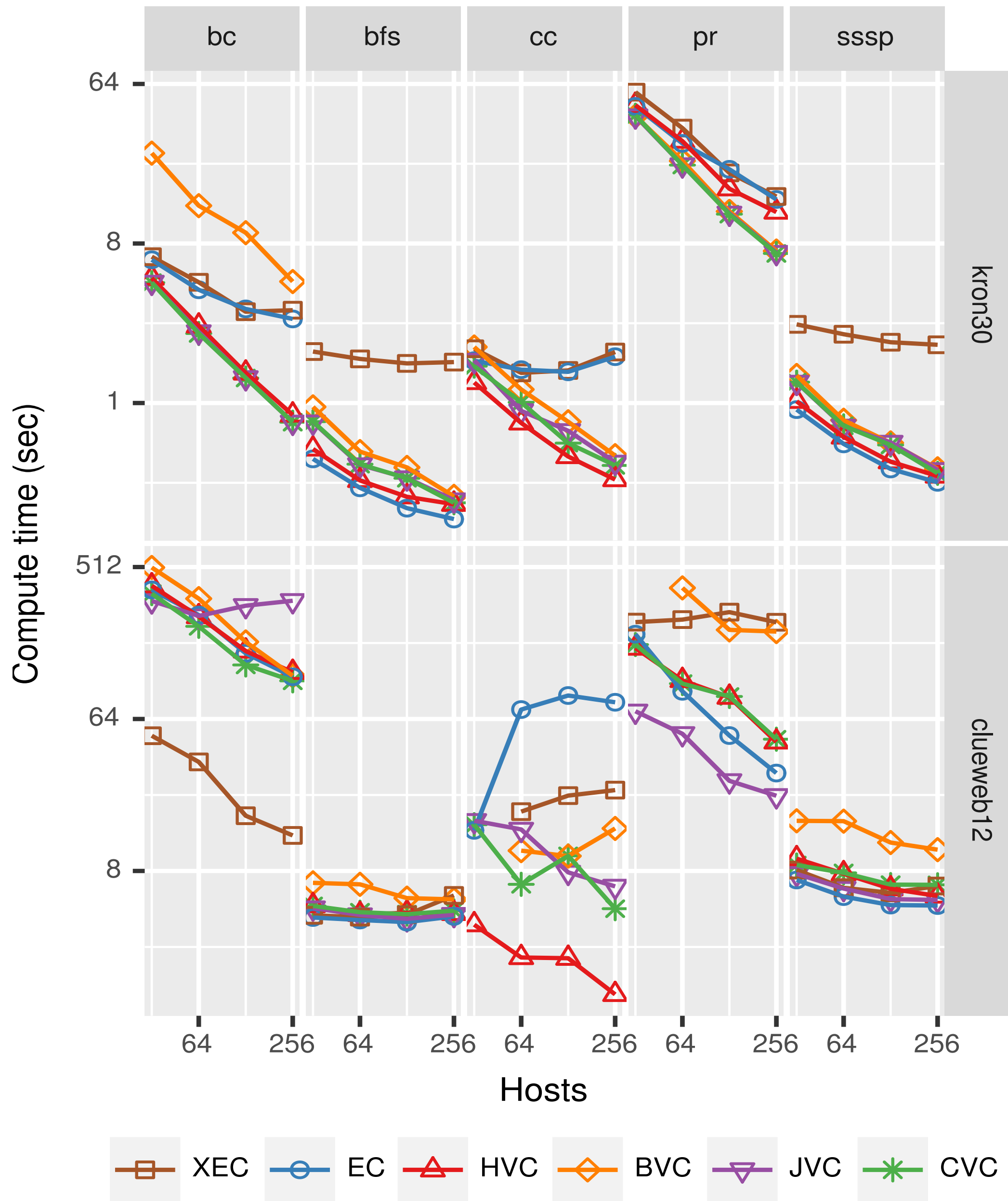


Figure 4.7: Compute time of D-Galois with different partitioning policies on KNL hosts.

4.6.5 Strong Scaling

We measure the time for computation of each round or iteration on each host and take the maximum across hosts. Summing up over iterations, we get the computation time. Figure 4.7 shows the strong scaling of execution time (left) and computation time (right) for kron30 and clueweb12 (most partitioning policies run out of memory for wdc12 on less than 256 hosts). Some general trends are apparent. At small scale on 32 hosts, EC performs fairly well for almost all the benchmarks and is comparable to the best, but as we go to higher number of hosts, EC, XEC, and HVC do not scale. On the other hand, all 2D partitioning policies scale relatively better for almost all benchmarks and inputs. Among them, JVC scales better than BVC, and CVC scales better than JVC. In most cases, CVC has the best performance at scale and scales best.

CVC does not scale well for bfs and sssp on clueweb12 due to computation time not scaling well. In all other cases, both computation time and execution time scale. For bfs and sssp, CVC is still better than the others in execution time since computation times of all policies do not scale. This is likely due to the computation being too small for it to scale (both execute more than 180 iterations, so each iteration has little to compute on each host at scale).

Computation time of all policies scales similarly in most cases. However, their execution time scaling differs. This is most evident in EC and CVC.

This indicates that the difference in the policies arises due to the communication. We analyze this next.

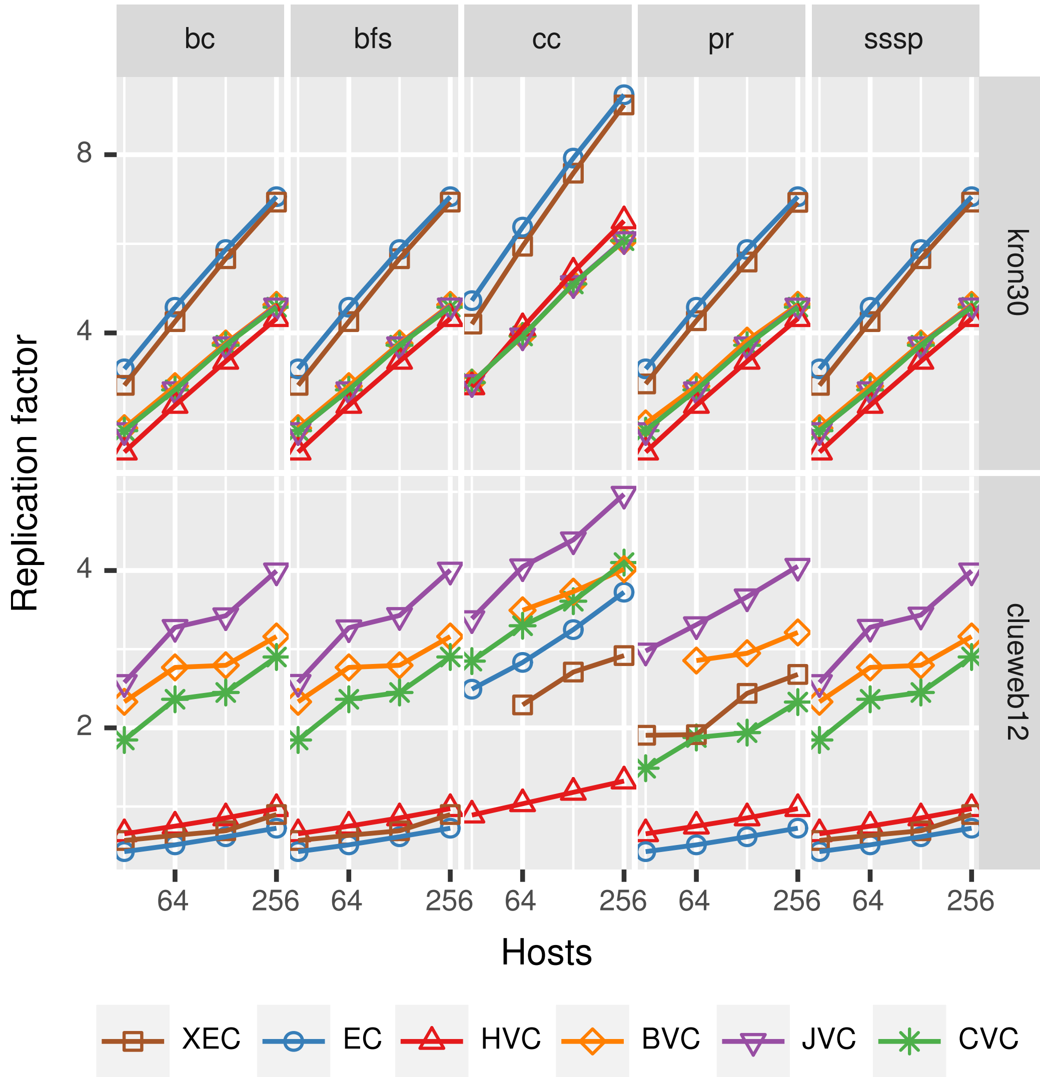


Figure 4.8: Replication factor of D-Galois with different partitioning policies on KNL hosts.

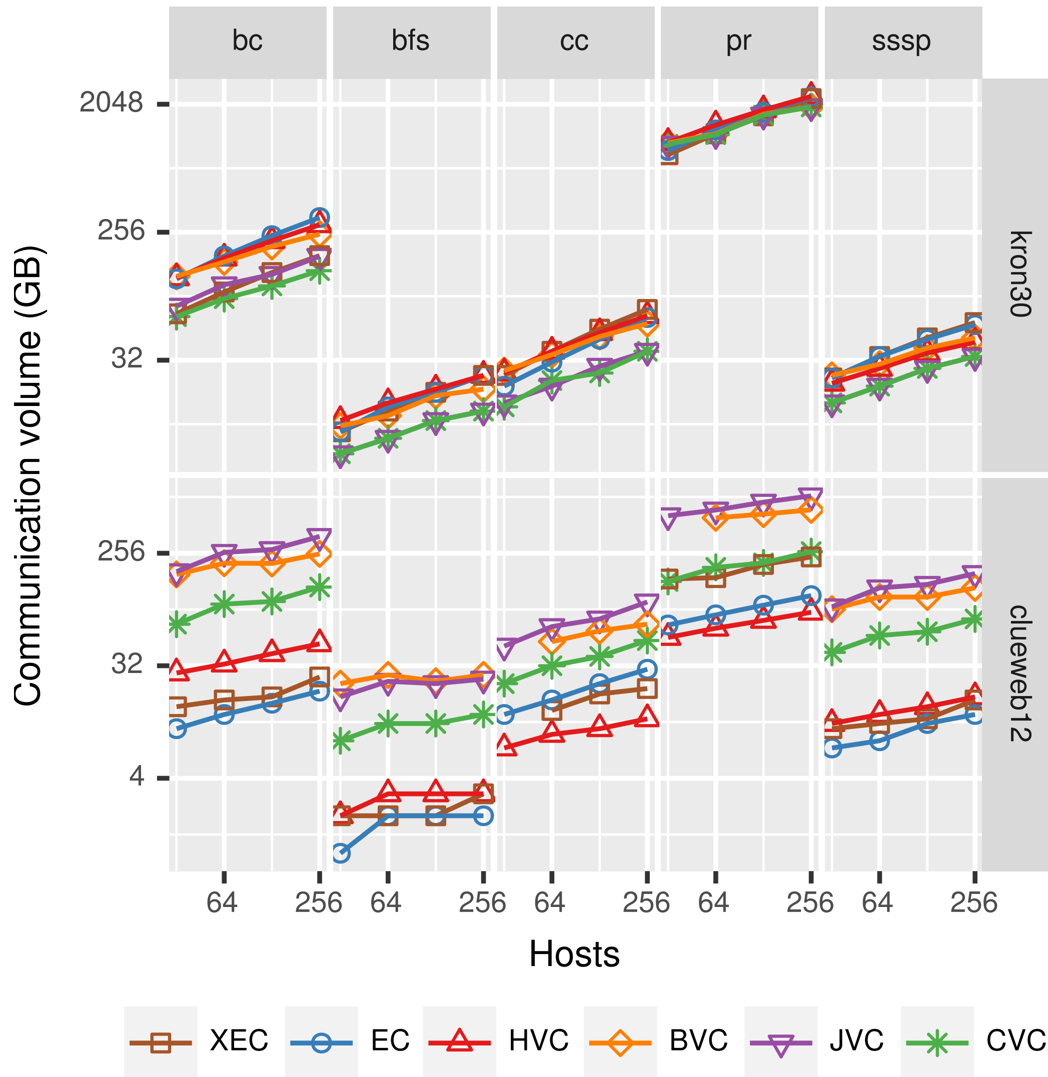


Figure 4.9: Communication volume of D-Galois with different partitioning policies on KNL hosts.

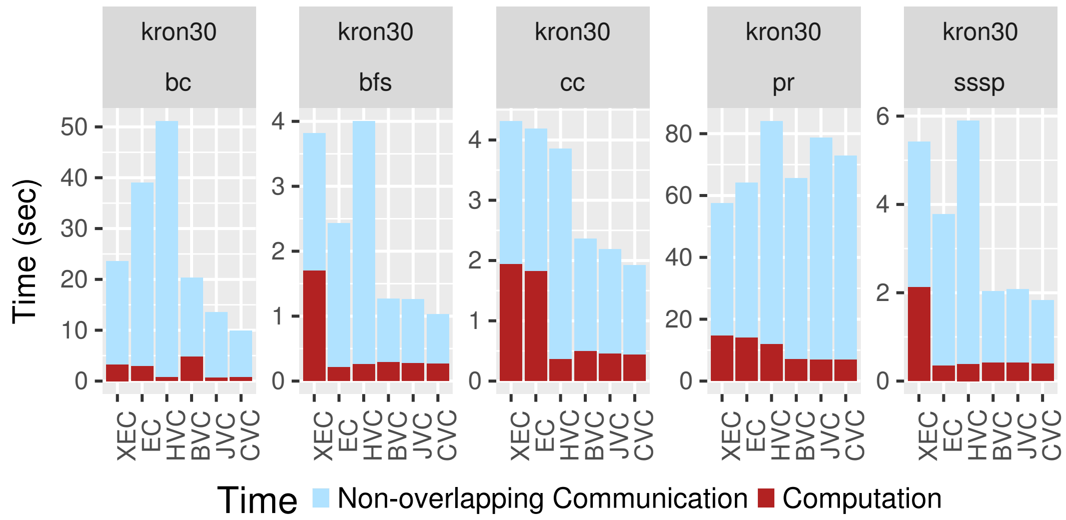


Figure 4.10: Breakdown of execution time on 256 hosts: kron30.

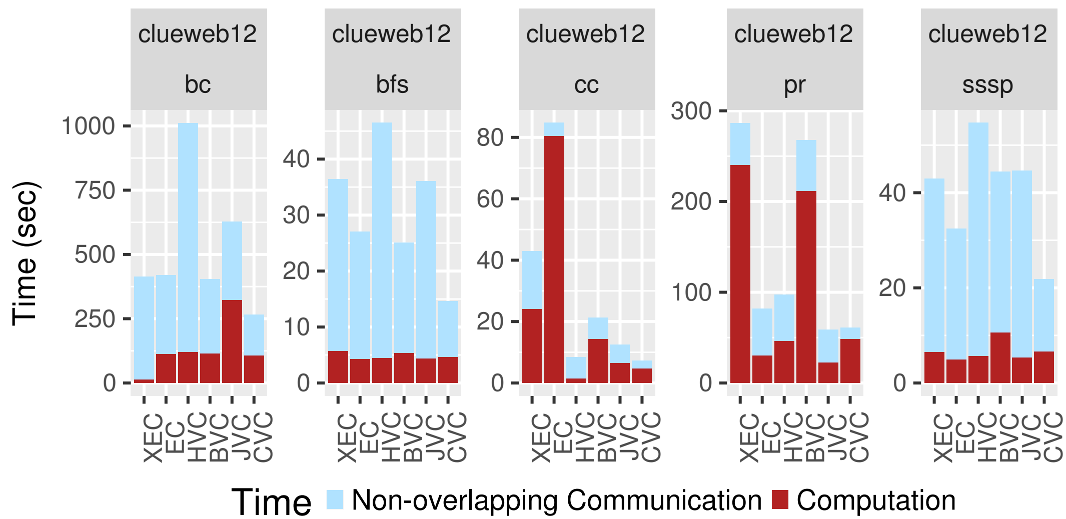


Figure 4.11: Breakdown of execution time on 256 hosts: clueweb12.

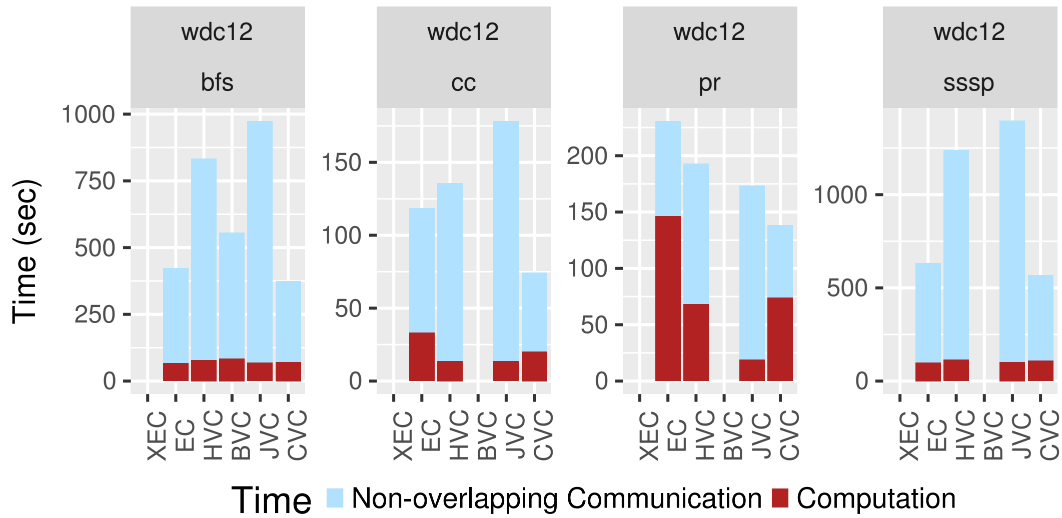


Figure 4.12: Breakdown of execution time on 256 hosts: wdc12 (XEC and BVC run out-of-memory).

4.6.6 Communication Analysis

The total communication volume is the sum of the number of bytes sent from one host to another during execution. Table 4.7 shows the replication factor and total communication volume for wdc12 on 256 hosts (XEC and BVC run out of memory). Figure 4.9 shows how the replication factor (left) and the total communication volume (right) scale as we increase the number of hosts for kron30 and clueweb12. The replication factor increases with the number of hosts for all partitioning policies as expected. Similarly, the total communication volume increases with the number of hosts as more data needs to be exchanged to synchronize those proxy nodes across hosts. However, the difference in replication factor across policies can vary from that of communication volume. This is most evident for kron30: although EC has a much

higher replication factor than the other policies, the communication volume of EC is close to that of others. This demonstrates that replication factor need not be the sole determinant for the communication volume.

For kron30, EC corresponds to IEC, so there are no updates sent from mirrors to masters; only masters send updates to mirrors. Vertex-Cut policies like HVC and CVC, however, send from mirrors to masters and then from masters to mirrors. Thus, if EC has roughly twice the replication factor of HVC, EC would still communicate the same volume as HVC. This can analyzed using our analytical model in Section 4.5.2. We estimate the communication volume for EC, HVC, and CVC using Equations 4.2, 4.4, and 4.5, respectively. The values to use for the replication factor $\overline{r}$ and the size of the data b are straightforward to determine. We assume that if a node is updated, all its mirrors are updated, so we use a value of 1 for f. These equations estimate the volume for a given round or iteration. To get the total communication volume, we can sum over all rounds. We instead replace the number of updates u in a round with the total number of updates performed on the graph across all the rounds to estimate the total communication volume.

Table 4.9 presents the estimated communication volume of different applications and policies for kron30 along with the replication factor and the observed communication volume. The estimated volume can be more than the observed volume because we assume f is 1, which is an over-approximation. The observed volume can be higher than the estimated volume as the estimation does not account for the metadata like node IDs communicated along

Table 4.9: Communication volume estimated by the model vs. observed communication volume on 128 KNL hosts for kron30.

		IEC	**HVC**	**CVC**
bfs	**Replication Factor**	5.53	**3.58**	3.81
	Estimated Volume(GB)	15.63	19.73	**2.82**
	Observed Volume(GB)	19.26	20.29	**11.69**
cc	**Replication Factor**	7.89	5.07	**4.84**
	Estimated Volume(GB)	30.07	71.02	**20.44**
	Observed Volume(GB)	44.51	50.40	**26.08**
pr	**Replication Factor**	5.53	**3.58**	3.81
	Estimated Volume(GB)	1153.56	1243	**932.06**
	Observed Volume(GB)	1797.30	1867.06	**1715.68**
sssp	**Replication Factor**	5.53	**3.58**	3.81
	Estimated Volume(GB)	36.97	40.60	**6.99**
	Observed Volume(GB)	44.63	35.55	**27.64**

with node values or updates, which is an under-approximation. Such approximations are fine because the relative ordering of the communication volume among different partitioning policies is important, not the absolute values. From the estimated volume for all the benchmarks, we can see that our analytical model predicts CVC to communicate the least amount of volume even if CVC has higher replication factor than HVC. A similar pattern can also be seen in the observed communication volume for all the benchmarks, where CVC has the minimum volume but not necessarily the minimum replication factor. This validates our analytical model, stating that the replication factor is not the sole determinant for communication volume.

In Figure 4.9, we see that CVC may have a higher replication factor and communication volume than the other policies, yet it performs better than the other policies in most cases. Figures 4.10, 4.11 and 4.12 show the breakdown

of execution time into computation time and non-overlapped communication time (the rest of the execution time) on 256 hosts. It is clear that CVC is doing better (except pagerank on kron30) because the communication time is lower. We can see that more communication volume does not always imply more communication time. For example, in pr and sssp on clueweb12, CVC has higher replication factor and more communication volume than EC and HVC but lower communication time. Figure 4.5 shows the percentage of rounds in those applications and policies that have low ($\leq$ 100MB), medium ($>$ 100MB and $\leq$ 1000MB), and high ($>$ 1000MB) communication volume. CVC increases the number of high volume rounds of pr and sssp over EC on both 32 and 256 hosts. Our micro-benchmarking in Section 4.5.3 shows that CVC yields significant speedups over EC on 256 hosts in both low and high volumes, which outweighs the increase in high volume rounds. In contrast, the increase in high volume rounds on 32 hosts for CVC over EC causes a slowdown in communication time since there is very little difference between CVC and EC at this scale for the same communication volume. This validates the claim that the communication time depends on both the communication volume and the communication pattern and shows that CVC has much less communication overhead than other policies at large scale.

4.7 Choosing a Partitioning Policy

Based on the results presented in Section 4.6, we present a decision tree (Figure 4.13) to help users choose the most suitable partitioning strategy

based on the following parameters:

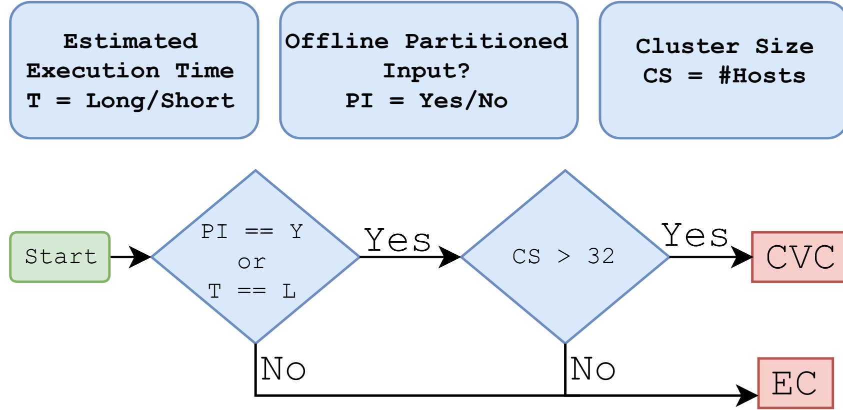


Figure 4.13: Decision tree to choose a partitioning policy.

1. Whether the input is partitioned offline: The time it takes to partition and load the graph (online) depends on the complexity of the partitioning strategy. EC takes least amount of time, whereas strategies like XEC [152] and CVC [23] take more time as they involve analysis and communication during partitioning. It makes sense to use complex partitioning strategies if the benefits gained from them outweigh the time spent in partitioning. D-Galois also supports direct loading of offline partitioned graphs, in which case partitioning time is not a factor as the partitioned graph is on disk.

2. Whether the execution time is estimated to be long or short: The amount of time spent in execution of the application plays a vital role in determining if it makes sense to invest time in a good partitioning strategy. Spending more time in partitioning makes more sense for long-running

Table 4.10: % difference in execution time (excluding partitioning time) on KNL hosts between the partitioning strategy chosen by the decision tree and the optimal one (wdc12 is omitted because chosen one is always optimal; 0% means that chosen one is optimal).

		32	**64**	**128**	**256**
kron30	**bc**	44.74%	0%	0%	0%
	bfs	13.33%	13.68%	0%	0%
	cc	12.5%	26.63%	0%	0%
	pr	9.31%	36.38%	30.63%	20.99%
	sssp	21.31%	23.26%	0%	0%
clueweb12	**bc**	68.98%	60.59%	21.32%	0%
	bfs	0%	37.24%	0%	0%
	cc	55.62%	52.16%	51.35%	0%
	pr	11.89%	18.69%	26.09%	3.62%
	sssp	0%	43.83%	17.33%	0%

applications as they involve more rounds of communication which can benefit from a well-chosen partitioning strategy. The user can easily identify whether the application is expected to run for a long time (e.g., by using algorithm complexity). Applications such as bc tend to be more complex as they have multiple phases within each round of computation, and they involve floating-point operations; therefore, they are expected to have longer execution times. On the other hand, applications like bfs are relatively less complex and can be classified as short-running applications.

3. Cluster size: Our results show that the performance of different partitioning strategies also depend on the number of hosts on which the application is run.

Table 4.11: % difference in execution time (excluding partitioning time) on Skylake hosts between the partitioning strategy chosen by the decision tree and the optimal one.

		8	256
kron30	**bc**	21.79%	0%
	bfs	0%	0%
	cc	0%	0%
	pr	0%	0%
	sssp	0%	0%
clueweb12	**bc**	0%	0%
	bfs	0%	0%
	cc	12.84%	0%
	pr	0%	0%
	sssp	0%	11.34%

Figure 4.13 illustrates our decision tree to choose a partitioning strategy. For short running applications, if the graph is not already partitioned, we recommend using simple EC. Additionally, if the graph is already partitioned and the number of hosts is less than 32, simple EC should suffice. For long-running applications, the decision to use EC or CVC primarily depends on the cluster size. If the number of machines is more than 32, it makes sense to invest partitioning time in CVC. Otherwise, EC is recommended.

Choosing the best partitioning strategy is a difficult problem as it depends on several factors such as properties of input graphs, applications, number of hosts (scale), etc. Therefore, the decision tree may not always suggest or choose the best partitioning strategy. Tables 4.10 and 4.11 illustrate this point by showing the percentage difference in the application execution time between the chosen partitioning strategy and the best-performing or optimal strategy

at different number of hosts assuming the input graphs used are already partitioned (i.e., graph construction time for all strategies are same). A value of zero means that the chosen partitioning strategy performs the best. In many cases, the chosen strategy performs best, particularly at 128 and 256 hosts. For kron30, this difference in most of the cases is under 20%. For clueweb12, the difference is slightly higher, especially for bc at 32 hosts, for which XEC performs best rather than simple EC (XEC uses a community detection technique for partitioning which provides compute locality for the compute-heavy bc algorithm). Nonetheless, the decision tree chooses a partitioning strategy that performs well in most cases.

4.8 Related Work

Several distributed-memory graph processing frameworks have been published in the past few years [33, 39, 43, 68, 70, 79, 80, 84, 91, 99, 106, 117, 170, 180, 181, 183, 195] These systems use graph partitioning to scale out computations on graphs or sparse matrices that do not fit in the memory of a single node. In the graph analytics literature, partitioning strategies are classified into Edge-Cuts [7, 88–90, 152, 154, 189, 195] and Vertex-Cuts [28, 39, 70, 92, 100, 129, 150, 163]. In the matrix literature, they are classified into 1D and 2D partitionings [23, 35]. 1D partitionings are equivalent to the class of Edge-Cuts, whereas 2D partitionings are strictly a sub-class of Vertex-Cuts as they are more restricted.

1D partitionings or Edge-Cuts: METIS [7, 89] and XtraPulp [152] par-

tition the graph based on connected components. XtraPulp has been shown to partition large graphs in a few minutes, but they do not compare against general Vertex-Cuts. Streaming Edge-Cut policies [154, 195] partition the graph in a pass or two over the edges. This work evaluates XtraPulp and edge-balanced Edge-Cuts to represent non-streaming and streaming Edge-Cuts, respectively.

2D partitionings: 2D partitionings have been studied in both dense matrix and sparse matrix communities [97]. CheckerBoard 2D partitioning (BVC) [35] is used in CombBLAS, a sparse matrix library. Jagged-like partitioning (JVC) [35] and Cartesian Vertex-Cut (CVC) [23] have been evaluated for generalized sparse matrix vector computation. However, these strategies have never been evaluated on work-efficient data-driven graph algorithms, and there are no comparisons with other policies like Hybrid Vertex-Cut [39].

Vertex-Cuts that are neither 1D nor 2D partitionings: PowerGraph [70] is the first graph analytical system to develop a streaming Vertex-Cut partitioning heuristic targeting power-law graphs. PowerLyra [39] proposed a streaming Vertex-Cut heuristic called Hybrid Vertex-Cut (HVC) that handles high-degree nodes differently from low-degree nodes. Bourse et al. [28] analyze balanced Vertex-Cut partitions theoretically and propose a least incremental cost (LIC) heuristic with approximation guarantees. Petroni et al. [129] proposed High-Degree (are) Replicated First (HDRF), a novel streaming Vertex-Cut graph partitioning algorithm that exploits skewed degree distributions by explicitly taking into account vertex degree in the placement decision. These papers do not compare their approaches to 2D block partitionings.

Studies of partitioning policies: There are several studies [6, 61, 100, 167] that have compared the impact of partitioning strategies on application execution time. Yun et al [61] compares various distributed graph analytics systems on different design aspects including graph distribution policies and concludes that the Vertex-Cut partitioning strategy always outperforms the Edge-Cut on vertex (neighbor-based) programs, which is not the case as shown by this work. LeBeane et al. [100] studies the impact of relative computational throughput of hosts in heterogeneous setting on various partitioning strategies for graph analytics workloads using PowerGraph. Verma et al. [167] evaluates different partitioning strategies provided by distributed graph analytics systems, namely, PowerGraph, GraphX, and PowerLyra, and it suggests the best partitioning strategy for each system among the strategies provided by that system. These studies were done at a very small scale of 10 to 25 hosts. Verma et al. [167] also compares various partitioning strategies on PowerLyra. However, PowerLyra does not optimize communication for the non-native partitioning strategies adopted from other systems. In a recent study, Abbas et al. [6] compares various streaming partitioning policies using a distributed runtime based on Apache Flink [34] and concludes that low-cut algorithms (with low replication factor) perform better for communication-intensive applications. However, the study was done on a small 17 host cluster, and the largest graph considered was Friendster, which easily fits in the memory of a single host in the cluster used in our study.

To the best of our knowledge, no previous study performs a quantitative

comparison of partitioning strategies with communication optimized for each partitioning strategy at scale for work-efficient graph analytics applications. In this work, we used D-Galois, an efficient distributed-memory graph processing system based on Gluon runtime [51] that optimizes communication specifically for each partitioning strategy, on 256 KNL hosts with a total of 69K threads. However, our study and observations are not limited to D-Galois, and they should generalize to other systems that optimize communication based on the partitioning strategy.

Chapter 5

Phoenix: A Substrate for Resilient Distributed Graph Analytics[1]

5.1 Motivation

For the distributed systems mentioned in Chapter 3 and 4, fault tolerance is an important concern for long-running graph analytics applications on large clusters. Some state-of-the-art high-performance graph analytics systems such as D-Galois [51] and Gemini [195] do not address fault tolerance; since the mean time between failures in medium-sized clusters is of the order of days [147], the approach taken by these systems is to minimize overheads for fault-free execution and restart the application if a fault occurs. Older distributed graph analytics systems rely on checkpointing [68, 70, 103, 106, 142, 148, 172], which adds overhead to execution even if there are no failures. Furthermore, all hosts have to be rolled back to the last saved checkpoint if even one host fails.

[1]This work was originally published in the proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 2019 [52]. The first and the second authors contributed equally to this work and were responsible for the key concepts, abstractions, and implementation of ideas for the resilient graph analytics in the distributed setting. Other co-authors helped with the experiments, data collection, and presentation.

More recent systems such as GraphX [181], Imitator [172], Zorro [136], and CoRAL [171] have emphasized the need to avoid taking checkpoints and/or to perform *confined recovery* in which surviving hosts do not have to be rolled back when a fault occurs. The solution strategies used in these systems are quite diverse and are described in Section 6.3. For example, GraphX is built on top of Spark [186], which supports a dataflow model of computation in which computations have transactional semantics and only failed computations need to be re-executed. Imitator requires an *a priori* bound on the number of faulty hosts, and it uses replication to tolerate failure, which constrains the size of graphs it can handle on a cluster of a given size. Zorro provides zero overhead in the absence of faults, but it may provide only approximate results if faults happen. CoRAL takes asynchronous checkpoints and performs confined recovery but it is applicable only to certain kinds of graph applications.

In this work, we present Phoenix, a communication and synchronization substrate to provide fail-stop fault tolerance for distributed-memory graph analytics applications. *The insight behind Phoenix is that to recover from faults, it is sufficient to restart the computation from a state that will ultimately produce the correct result.* These states, called *valid* states in this work, are defined formally in Section 5.4. All states that are reached during fault-free execution are valid states, but in general, there are valid states that are not reachable during fault-free execution. On failure, Phoenix sets the state of revived hosts appropriately with the aid of the application programmer so that the global state is valid and continues execution, thus performing confined

recovery in which surviving hosts do not lose progress.

5.2 Contributions

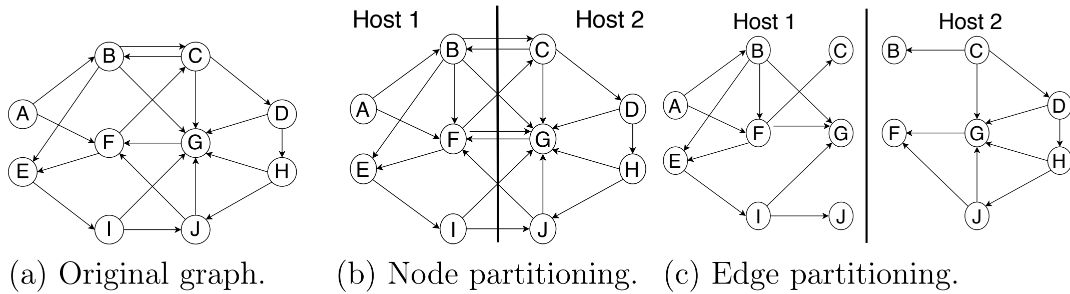
This work makes the following contributions.

- We present Phoenix, a substrate that can be used to achieve fault tolerance to fail-stop faults for distributed graph analytics applications without any observable overhead in the absence of faults. Phoenix does not use checkpoints, but it can be integrated seamlessly with global and local checkpointing.

- We describe properties of graph analytics algorithms that can be exploited to recover efficiently from faults and classify these algorithms into several categories based on these properties (Section 5.4). The key notions of *valid* states and *globally consistent* states are also introduced in this section.

- We describe techniques for recovering from fail-stop faults without losing progress of surviving hosts for the different classes of algorithms and show how they can be implemented by application developers using the Phoenix API (Section 5.5).

- We show that D-Galois [51], the state-of-the-art distributed graph analytics system, can be made fault tolerant without performance degradation by using Phoenix while outperforming GraphX [181], a fault tolerant system, and Gemini [195], a fault intolerant system, by a geometric



(a) Original graph. (b) Node partitioning. (c) Edge partitioning.

Figure 5.1: An example of partitioning a graph for two hosts in two different ways.

mean factor of $\sim 24\times$ and $\sim 4\times$, respectively. In addition, when faults occur, Phoenix generally outperforms the traditional checkpoint-restart technique implemented in D-Galois (Section 5.6).

5.3 Background

Background and prior work on fault tolerance in these systems is summarized in the following Section 5.3.1. Phoenix's programming and execution models are same as described in Chapter 1 Section 1.1.1 and Section 1.1.2 respectively. Figure 5.1 shows a sample graph and its partitioning across hosts and will be used as a running example throughout this chapter.

5.3.1 Fault Tolerant Distributed Graph Analytics

A fail-stop fault in a distributed system occurs when a host in the cluster fails without corrupting data on other hosts. Such faults can be handled with a *rebirth-based approach*, in which the failed host is replaced with an unused host in the cluster, or a *migration-based approach* [149], which dis-

tributes the failed host's partition among the surviving hosts. We describe and evaluate Phoenix using rebirth-based recovery, but it can also be used with migration-based recovery.

Several fault tolerant distributed graph analytics systems rely on checkpointing [68, 70, 103, 106, 142, 148, 172]: the state of the computation is saved periodically on stable storage by taking a globally consistent snapshot [37], and when a fail-stop fault is detected, the computation is restarted from the last checkpoint [185]. One disadvantage of checkpointing is that every host has to be rolled back to the last checkpoint even if only one host fails. Some existing distributed graph analytics systems have devised ways to avoid taking global checkpoints or rolling back live hosts, as described below.

GraphX [181] is built on Spark [186], and achieves fault tolerance by leveraging Resilient Distributed Datasets (RDD) from Spark to store information on how to reconstruct graph states in the event of failure; if the reconstruction information becomes too large, it checkpoints the graph state. GraphX is very general and can recover from failure of any number of hosts, but unlike Phoenix, it does not exploit semantic properties of graph applications to reduce overhead.

Table 5.1 compares the performance of GraphX and Phoenix on 32 hosts of the Wrangler cluster (experimental setup described in Section 5.6). The table also shows the performance of D-Galois [51] and Gemini [195], which are state-of-the-art distributed graph analytics systems that do not support fault tolerance. The Phoenix system in Table 5.1 is D-Galois made resilient by

Table 5.1: *Fault-free execution on 32 hosts of Wrangler.*

App	Input	Total Time (s)				Phoenix Speedup	
		GraphX	Gemini	D-Galois	Phoenix	GraphX	Gemini
cc	**twitter50**	110.8	29.0	8.2	8.2	13.5	3.5
	rmat28	166.5	80.0	20.6	20.6	8.1	3.9
	kron30	1538.2	347.1	56.2	56.2	27.4	6.2
pr	**twitter50**	2111.6	75.1	31.6	31.6	66.7	2.4
	rmat28	5355.5	180.9	47.7	47.7	112.3	3.8
	kron30	797.6	426.4	78.9	78.9	10.1	5.4
sssp	**twitter50**	153.1	24.2	8.5	8.5	18.0	2.8
	rmat28	158.1	77.7	11.1	11.1	14.3	7.0
	kron30	1893.2	346.7	46.0	46.0	41.2	7.5

using the Phoenix substrate. *In fault-free execution, Phoenix performance is the same as D-Galois, and the system does not have any observable overhead.* Phoenix programs also run on average $\sim 4\times$ faster than the same programs in Gemini. *In addition, Phoenix programs are $\sim 24\times$ faster, on average, than GraphX programs in the absence of faults while providing the same level of fault tolerance.*

Imitator [172] is designed to tolerate a given number of faults: if n-way fault tolerance is desired, the system ensures that at least $n+1$ proxies are created for every graph node, and it updates the labels of all these proxies at every round to ensure that at least one proxy survives simultaneous host failure. The memory requirements of Imitator grow proportionately with n, and keeping these additional proxies synchronized adds overhead even when there are no failures. As shown in [172], the ability to tolerate a single fault results in an overhead of $\sim 4\%$ even if no faults occur; similarly, tolerating 3 faults increases the overhead to $\sim 8\%$. In contrast, Phoenix does not require

an *a priori* bound on the number of simultaneous faults, and it does not require creating any more proxies than are introduced by graph partitioning.

Like Imitator, Zorro [136] relies on node proxies for recovery, but it only uses the proxies created by graph partitioning. If no proxies of a given node survive failure, execution continues but may produce an incorrect answer. An advantage of Zorro is that it does not have any overhead in the absence of failures, unlike GraphX and Imitator in which fault tolerance comes at the cost of overhead even when no failures occur. Phoenix shares this advantage with Zorro. While both Phoenix and Zorro incur overhead when failures happen, Phoenix is guaranteed to produce the correct answer unlike Zorro. Specifically, for self-stabilizing and locally-correcting algorithms (defined in Section 5.4), Zorro will yield the correct result if run till convergence[2]. However, for globally-correcting algorithms, Zorro will yield an incorrect result even if run till convergence. With respect to Zorro, our contributions are (1) defining the different classes of algorithms and (2) designing recovery mechanisms, specific to that class, that yield the correct result. For self-stabilizing algorithms, Phoenix uses a simpler recovery mechanism than Zorro. For locally-correcting algorithms, the recovery mechanisms of Phoenix is similar to that of Zorro. For globally-correcting algorithms, Phoenix uses a novel recovery approach with the help of the programmer.

[2]The algorithms evaluated in Zorro [136] are self-stabilizing or locally-correcting. They were only run for a fixed number of total iterations when faults occur and a loss of precision was reported. However, if they were run until convergence, they would have been precise.

CoRAL [171] is based on asynchronous checkpointing [108, 157]: hosts take local checkpoints independently, so no global coordination is required for checkpointing. Nonetheless, it incurs overhead during fault-free execution. CoRAL is applicable only to a subset of the algorithms that can be handled by Phoenix. Specifically, CoRAL can handle only self-stabilizing or locally-correcting algorithms (defined in Section 5.4). In CoRAL, like in other systems based on confined recovery [106, 148], the surviving hosts wait for the revived hosts to recover their lost state before continuing execution. Phoenix, in contrast, performs confined recovery without waiting for the revived hosts to recover their lost state.

5.4 Classes of Graph Analytics Algorithms

This section describes the key properties of graph analytics algorithms that are exploited by Phoenix. Although Phoenix handles fail-stop faults in distributed-memory clusters, these algorithmic properties are easier to understand in the context of data corruption errors[3] [153] (or transient soft faults) in shared-memory programming, so we introduce them in that context. Section 5.4.1 introduces the key notions of *valid* and *globally consistent* states. Section 5.4.2 shows how these concepts can be used to classify graph analytics algorithms into a small number of categories. Section 5.4.3 shows how this classification is applicable to a distributed-memory setting.

[3] Data corruption means some bits in the data are flipped. In this work, we consider fail-stop faults in distributed-memory and data corruption errors in shared-memory. We do not consider data corruption in distributed-memory.

5.4.1 Overview

In shared-memory execution, the state of the computation can be described compactly by a vector of node labels in which there is one entry for each node. If s is such a vector and v is a node in the graph, let $s(v)$ refer to the label of node v in state s. For example, if the initial state in a breadth-first search (bfs) computation is denoted by s_i and the source is node r, then $s_i(r) = 0$ and $s_i(v) = \infty$ for all other nodes v in the graph. During the computation, these labels are updated by applying the relaxation operator to nodes in the graph to change the state. When the algorithm terminates, the final state s_f will contain the bfs labels of all nodes. Therefore, the evolution of the state can be viewed as a trajectory beginning at s_i and ending at s_f in the set S of all states. In general, there are many such trajectories, and since the scheduler is permitted to make non-deterministic choices in the order of processing nodes, different executions of a given program for a given input graph may follow different trajectories from s_i to s_f. Figure 5.2 shows a trajectory for an application.

We define these concepts formally below.

Definition 5.4.1. For a given graph analytics program and input graph, let S be the set of states, and let s_i and s_f denote the initial and final states, respectively.

Definition 5.4.2. For $s_m, s_n \in S$, s_n is said to be a *successor state* of s_m if applying the operator to a node when the computation is in state s_m changes the state to s_n.

A *trajectory* is a sequence of states $s_0, s_1, ..., s_l$ such that for all $0 \leq j < l$, s_{j+1} is a successor state of s_j.

Two subsets of the set of states, which we call *valid* states (S_V) and *globally consistent* states (S_{GC}), are of interest.

Definition 5.4.3. A state s_g is *globally consistent* if there is a trajectory $s_i, ..., s_g$ from the initial state s_i to s_g. Denote S_{GC} the subset of globally consistent states in S.

A state s_v is *valid* if there is a trajectory $s_v, ..., s_f$ from s_v to the final state s_f. Denote S_V the subset of valid states in S.

Intuitively, a state is globally consistent if it is "reachable" (along some trajectory) from the initial state, and a state is valid if the final state (the "answer") is reachable from that state [57, 77, 123, 144, 145]. Every globally consistent state is valid; otherwise, there is a state s reachable from s_i from which s_f is not reachable, which is impossible. Therefore,

$$S_{GC} \subseteq S_V \subseteq S \tag{5.1}$$

In general, both containments can be strict, so there can be valid states that are not globally consistent, and there can be states that are not valid. This can be illustrated with bfs. Consider a state s_x in which the label of the root is 0, the labels of its immediate neighbors are 2, and all other node labels are ∞. This is not a state reachable from s_i, so $s_x \notin S_{GC}$. However, it is a valid state since applying the operator to the root node changes the labels of

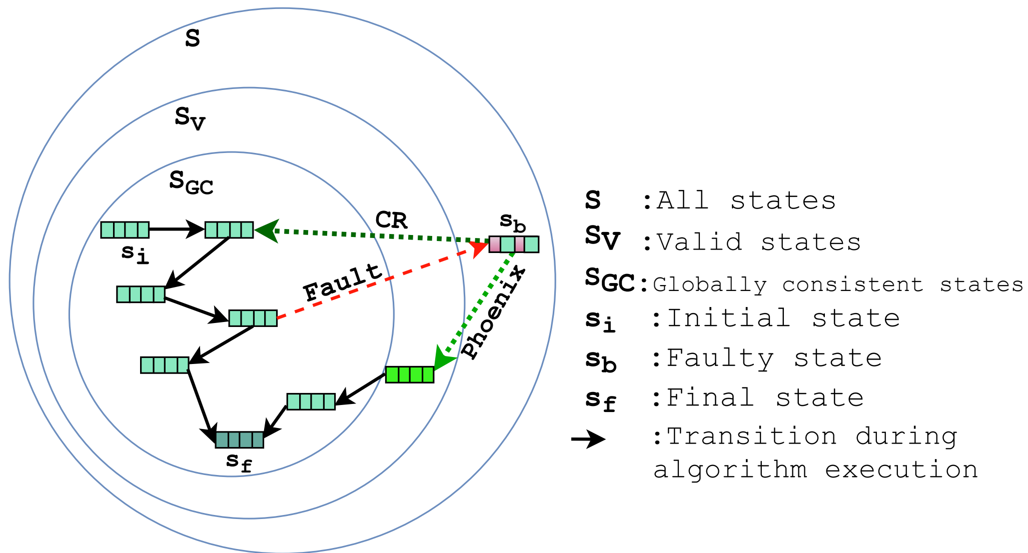


Figure 5.2: States during algorithm execution and recovery.

the neighbors of the root to their correct (and final) values. This shows that $S_{GC} \subset S_V$ in general. To show that not all states are valid, consider the state s_z where $s_z(v) = 0$ for all nodes v. No other state is reachable from this state; in particular, s_f is not reachable, so s_z is not a valid state. Therefore, $S_V \subset S$ in general.

The recovery approach used in Phoenix can be explained using the notions of valid states and globally consistent states. Consider Figure 5.2, which shows a trajectory of states from the initial state s_i to the final state s_f in fault-free execution. Every intermediate state in the trajectory between s_i and s_f is a globally consistent state. Checkpointing schemes save (some) globally consistent states on stable storage and recover from a fault by restoring the last globally consistent state saved. Figure 5.2 shows this pictorially: s_b is an invalid state and a checkpoint-restart scheme, denoted by CR, recovers

by restoring the state to a globally consistent state. In contrast, as illustrated in Figure 5.2, *the key insight in Phoenix is that for recovery, we can restart the computation from a valid state that is not necessarily a globally consistent state.*

5.4.2 Classification of Graph Algorithms

The way in which Phoenix generates a valid state for recovery depends on the structure of the algorithm. There are four cases, discussed below in increasing order of complexity.

Self-stabilizing graph algorithms: All states are valid.

$$S_{GC} \subset S_V = S \tag{5.2}$$

Examples are topology-driven algorithms for collaborative filtering using stochastic gradient descent (cf), belief-propagation, pull-style pagerank, and pull-style graph coloring. In these algorithms, node labels are initialized to random values at the start of the computation, and the algorithm converges regardless of what those values are. Therefore, every state is valid, and no correction of the state is required when a data corruption error is detected in the shared-memory case.

Locally-correcting graph algorithms: The set of valid states is a proper superset of the set of globally consistent states and a proper subset of the set

of all states. In addition, each node v has a set of valid values L_v, and the set of valid states S_V is the Cartesian product of these sets.

$$
\begin{aligned}
S_{GC} \subset S_V \subset S \\
S_V = L_1 \times L_2 \times ... \times L_N
\end{aligned}
\tag{5.3}
$$

Some examples are breadth-first search (bfs), single-source shortest path (sssp), connected components using label propagation (cc), data-driven pagerank, and topology-driven k-core. For example, in bfs, all values are valid for the root node since the operator sets its label to 0. Therefore, $L_r = [0, \infty]$. For an immediate neighbor v of the root node, $L_v = [1, \infty]$, and so on for the neighbors of those nodes. Data corruption in the shared-memory case can be handled by setting the label of each corrupted node to ∞; the labels of all other nodes can remain unchanged. This may produce a state that is not globally consistent, but the properties of the algorithm guarantee that the final state is reachable from this valid state.

Globally-correcting graph algorithms: Valid states are distinct from globally consistent states and from the set of all states, but unlike in the previous case, validity of a state depends on some global condition on the labels of all nodes and cannot be reduced to the Cartesian product of valid values for individual node labels.

These algorithms are usually more work-efficient than their equivalent locally-correcting counterparts. Some examples are residual-based data-driven

pagerank (pr), data-driven k-core (kcore), and latent Dirichlet allocation. In these algorithms, the label of a node is dependent not only on the current labels of its neighbors but also on the history or change of these labels. Hence, to recover from data corruption errors in shared-memory, re-initializing the label of each corrupted node is insufficient. After re-initializing the corrupted nodes, all nodes can re-compute their labels using only the current labels of their neighbors. This restores a valid state from which the work-efficient algorithm would reach the final state.

Globally-consistent graph algorithms: Only a globally consistent state is a valid state.

$$S_{GC} = S_V \subset S \tag{5.4}$$

Betweenness centrality [76] is an example. Globally consistent snapshots [37] are required for recovery. In such cases, Phoenix may be used along with traditional checkpointing. We list this class only for the sake of completeness, and we do not discuss this class of algorithms further in this chapter.

5.4.3 Distributed Graph Analytics

The concepts introduced in this section for shared-memory can be applied to distributed-memory implementations of graph analytics algorithms as follows. The main difference from the shared-memory case is that a given node in the graph can have proxies on several hosts that can be updated independently during the computation. In BSP-style execution, all proxies of a given

node are synchronized at the end of every round, and at that point, they all have the same labels. Therefore, in the distributed-memory case, we consider a *round* to constitute one step of computation, and the global state at the end of each round is simply the vector containing the labels of the nodes at that point in time.

Definition 5.4.4. For $s_m, s_n \in S$, s_n is said to be a *successor state* of s_m if a single BSP round transforms state s_m to state s_n.

A *trajectory* is a sequence of states $s_0, s_1, ..., s_l$ such that for all $0 \leq j < l$, s_{j+1} is a successor state of s_j.

Globally consistent states and valid states can be defined as in Definition 5.4.3, and the classification of graph algorithms in Section 5.4.2 can now be used in the context of distributed-memory implementations.

5.5 Fault Tolerance in Distributed Memory

In this section, we describe how Phoenix performs confined recovery when a fault occurs without incurring overhead during fault-free execution. Section 5.5.1 presents Phoenix in the context of distributed execution of graph analytics applications. Sections 5.5.2, 5.5.3, and 5.5.4 illustrate Phoenix's recovery mechanisms for the three different classes of algorithms introduced in Section 5.4. Section 5.5.5 summarizes the Phoenix API and discusses how programmers can use it when writing applications.

```
Input  : Partition G_h = (V_h, E_h) of graph
         G = (V, E)
Output: A set of colors s(v) ∀v ∈ V
Let t(v) ∀v ∈ V be a set of temporary colors
 Function Init(v, s):
 |  s(v) = random color
foreach v ∈ V_h do
 |  Init(v, s)
end
repeat
 |  foreach v ∈ V_h do
 |   |  t(v) = s(v)
 |  end
 |  foreach v ∈ V_h do
 |   |  foreach u ∈ adj(v) and u < v do
 |   |   |  nc = nc ∪ {t(u)}
 |   |  end
 |   |  s(v) = smallest c such that c ∉ nc
 |   |
 |  end
 |  while Runtime.Sync(s) == Failed do
 |   |  Phoenix.Recover(Init, s)
 |  end
until ∀v ∈ V, s(v) = t(v);
```

algorithm 1: Greedy graph coloring (self-stabilizing).

```
Function Recover(Init, s):
 |  if h ∈ failed hosts H_f then
 |   |  foreach v ∈ V_h do
 |   |   |  Init(v, s)
 |   |  end
 |  end
```

algorithm 2: Phoenix API for self-stabilizing algorithms.

5.5.1 Overview

Fail-stop faults are detected and handled during the synchronization phase of a BSP round. Once a fail-stop fault is detected, the program passes the control to Phoenix using the Phoenix API. Phoenix reloads graph partitions from stable storage on the revived hosts that replace the failed hosts. Some of the nodes on these hosts may have proxies on surviving or healthy hosts, and if so, their state can be recovered from their proxies. In Phoenix, this is done using a minor variation of the synchronization call used to rec-

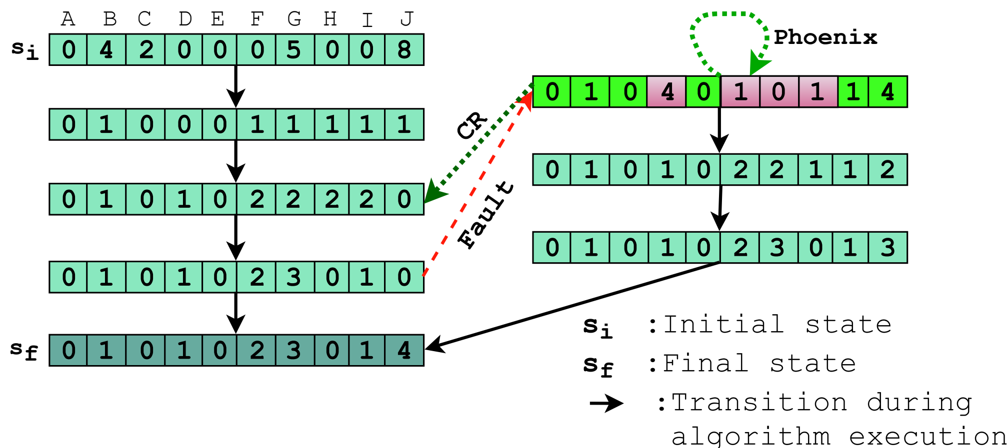


Figure 5.3: States of the graph in Figure 5.1(a), treated as an undirected graph, during the execution of greedy graph coloring.

oncile proxies during fault-free or normal execution. If the entire state can be recovered in this way, execution continues. However, if there are nodes for which no proxy exists on a healthy host, Phoenix restores the global state to a valid state using the approach described in detail in this section.

All algorithms presented use a generic interface called `Sync` to synchronize the state of all proxies of all nodes. Synchronization of different proxies of a node involves an *all-reduce* operation, and different systems support this in different ways. For example, the interface maps directly to Gluon's [51] `Sync` interface used in D-Galois. In Gather-Apply-Scatter (GAS) models like PowerGraph [70], the interface can be implemented using gather and scatter. During normal execution, `Sync` only synchronizes proxies that have been updated. In contrast, proxies of all nodes are synchronized during Phoenix recovery. The *reduction* operation for the same field could be different during normal execution and Phoenix recovery (e.g., addition in normal execution corresponds to

maximum during recovery). In addition, Gluon exploits the structural invariants of partitioning policies to avoid performing an *all-reduce* during normal execution, but we do not use this during Phoenix recovery. We omit these variations of `Sync` in the algorithms presented for the sake of simplicity.

Algorithms for each class are presented in pseudocode since they can be implemented in different programming models or runtimes. Each host h executes the algorithm on its partition of the graph. The work-list, when used, is local to the host. The `foreach` loop denotes a parallel loop that can be executed on multiple threads, using fine-grained synchronization to update the local state. For example, the loop can be implemented using the `do_all` construct in D-Galois with atomic updates to the local state. All algorithms use the generic interface, `Sync` or `SyncW`, to synchronize the state of all proxies; `SyncW` is similar to `Sync`, but it also adds nodes whose state is updated locally during synchronization to the work-list. Without the explicit synchronization or Phoenix calls, these algorithms can be executed on shared-memory systems. `Sync` and `SyncW` fail when at least one host has crashed, and when that occurs, the algorithms call a generic API for Phoenix that recovers from the failure.

The Phoenix API calls `Sync` at the end of its recovery. In the algorithms presented, we push this `Sync` call to the user code and omit the loading of partitions on the hosts replacing the crashed hosts. If the failures cascade, i.e., if failures occur during recovery, then the `Sync` called at the end of Phoenix's recovery will fail and Phoenix's recovery is re-initiated after partitions are loaded on the hosts replacing the failed hosts.

5.5.2 Self-Stabilizing Graph Algorithms

To illustrate how self-stabilizing algorithms are handled by Phoenix, Algorithm 1 shows greedy graph coloring for an undirected or symmetric graph. Every node has a label to denote its color which is initialized randomly. Nodes and colors are ordered according to some ranking function. The algorithm is executed in rounds. In each round, every node picks the smallest color that was not picked by any of its smaller neighbors in the previous round. The algorithm terminates when color assignments do not change in a round.

The programmer calls the Phoenix API (line 14) if `Sync` fails. Figure 5.3 shows possible state transitions after each round of algorithm execution for the graph in Figure 5.1(a). In this figure, colors are encoded as integers. Suppose that faults are detected in the fourth round when performing the `Sync` operation. To recover, Phoenix will initialize the colors of the proxies of the nodes on each failed host to a random color using the programmer supplied function. Some of these nodes might have proxies on healthy hosts, which will be recovered by the subsequent `Sync` call (line 13). The nodes which do not have any proxies on healthy hosts are shaded red in Figure 5.3. Algorithm execution then continues, converging in three more rounds. Phoenix supports such confined recovery by providing a thin API defined in Algorithm 2. Phoenix recovers a valid state for all self-stabilizing algorithms in this way because any state is a valid state (Equation 5.2).

```
Input  : Partition G_h = (V_h, E_h) of graph G = (V, E)
Input  : Source v_s
Output: A set of distances s(v) ∀v ∈ V
Function InitW(v, s, W_n):
    if v == v_s then
        s(v) = 0 ; W_n = W_n ∪ {v}
    else
        s(v) = ∞
    end
foreach v ∈ V_h do
    InitW(v, s, W_n)
end
repeat
    W_o = W_n ; W_n = ∅
    foreach v ∈ W_o do
        foreach u ∈ outgoing_adj(v) do
            if s(u) > s(v) + 1 then
                s(u) = s(v)+1 ; W_n = W_n∪{u}
            end
        end
    end
    while Runtime.SyncW(s, W_n) == Failed do
        Phoenix.Recover(InitW, s, W_n)
    end
until global_termination;
```

algorithm 3: Breadth first search (locally-correcting).

```
Function Recover(InitW, s, W_n):
    if h ∈ failed hosts H_f then
        foreach v ∈ V_h do
            InitW(v, s, W_n)
        end
    end
```

algorithm 4: Phoenix API for locally-correcting algorithms.

Figure 5.4: Breadth first search (locally-correcting).

5.5.3 Locally-Correcting Graph Algorithms

We use data-driven breadth-first search (bfs), shown in Algorithm 3, to explain how locally-correcting algorithms are handled by Phoenix. Every node has a label that denotes its distance from the source, which is initialized to 0 for the source and ∞ for every other node. The algorithm is executed in rounds, and in each round, the relaxation operator is applied to nodes on the bfs frontier, which is tracked by work-lists. Initially, only the source is in the

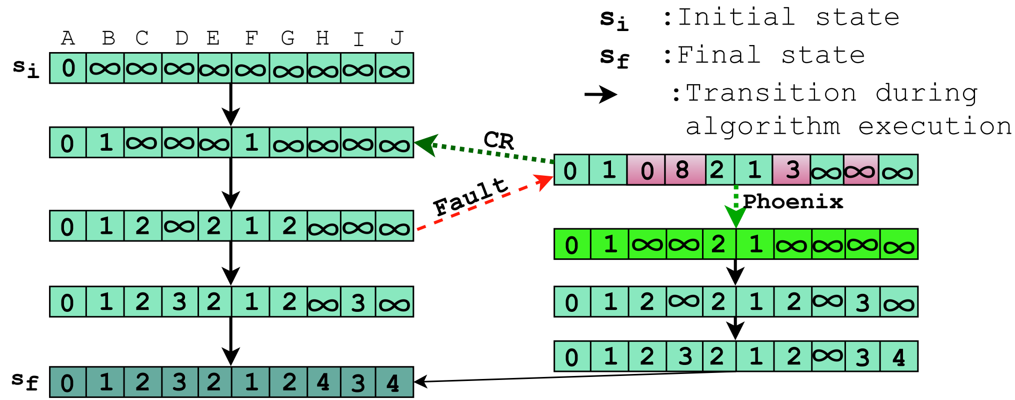


Figure 5.5: State of the graph in Figure 5.1(a) during the execution of data-driven breadth first search (locally-correcting).

work-list. If the neighbor's distance changes when relaxed, then it is added to the work-list for the next round. The algorithm terminates when there are no nodes in the work-list on any host.

The program calls the Phoenix API (line 15) when `SyncW` fails. When a fault occurs, Phoenix uses the given function to initialize the labels of the proxies on the failed hosts and update the work-list. Some of the nodes on the failed host may have proxies on healthy hosts, which are recovered in the subsequent `SyncW` call (line 14). This call also adds the proxies whose labels were recovered to the work-list. For all locally-correcting algorithms, the state is now valid because (i) the initial label of a node is a valid value (Equation 5.3) and (ii) the work-list ensures that all values lost will be recovered eventually. Phoenix supports such confined recovery by providing a thin API defined in Algorithm 4.

Figure 5.5 illustrates this for the graph in Figure 5.1(a). Hosts fail

```
Input  : Partition G_h = (V_h, E_h) of graph G = (V, E)
Input  : k
Output: A set of flags and degrees s(v) ∀v ∈ V
Function DecrementDegree(v, s):
    foreach u ∈ outgoing_adj(v) do
        s(u).degree = s(u).degree − 1
    end
Function ReInitW(v, s, W_n):
    s(v).degree = |outgoing_adj(v)|
    W_n = W_n ∪ {v}

Function InitW(v, s, W_n):
    s(v).flag = True
    ReInitW(v, s, W_n)
Function ComputeW(v, s, W_n):
    if s(v).degree < k then
        s(v).flag = False
        DecrementDegree(v, s)
    else
        W_n = W_n ∪ {v}

    end
Function ReComputeW(v, s, W_n):
    if s(v).flag = False then
        DecrementDegree(v, s)
    else
        W_n = W_n ∪ {v}

    end
foreach v ∈ V_h do
    InitW(v, s, W_n)
end
repeat
    W_o = W_n ; W_n = ∅
    foreach v ∈ W_o do
        ComputeW(v, s, W_n)
    end
    while Runtime.SyncW(s, W_n) == Failed do
        Phoenix.Recover(InitW, ReInitW, ReComputeW, s, W_n)
    end
until global_termination;
```

algorithm 5: Data-driven k-core (global-correcting).

```
Function Recover(InitW, ReInitW, ReComputeW, s, W_n):
    if h ∈ failed hosts H_f then
        foreach v ∈ V_h do
            InitW(v, s, W_n)
        end
    else
        foreach v ∈ V_h do
            ReInitW(v, s, W_n)
        end
    end
    if Runtime.SyncW(s, W_n) == Failed then
        return
    end
    W_o = W_n ; W_n = ∅
    foreach v ∈ W_o do
        ReComputeW(v, s, W_n)
    end
```

algorithm 6: Phoenix API for globally-correcting algorithms.

during the third round. CR would restore the state to that at the end of round 2, and this will need three more rounds to converge. Consider the nodes on the failed hosts that do not have proxies on the healthy hosts (shaded in red). Phoenix will initialize the distances of their proxies to ∞ since none of them is the source. The incoming neighbors of these nodes that have proxies on the healthy hosts will be recovered by the subsequent `SyncW` call and added to the work-list. Phoenix will resume execution and converge in three more rounds.

5.5.4 Globally-Correcting Graph Algorithms

To illustrate how Phoenix handles globally-correcting graph algorithms, we use degree-decrementing, data-driven k-core (kcore), shown in Algorithm 5. The k-core problem is to find the sub-graphs of an undirected or symmetric graph in which each node has degree at least k. Most k-core algorithms execute by removing nodes with fewer than k neighbors in the graph since these nodes cannot be part of a k-core. The removal of these nodes lowers the degrees of other nodes, which may enable more nodes to be removed from the graph.

Since explicitly deleting nodes and edges from the graph is expensive, implementations usually just mark a node as dead and decrement the degree of its neighbors. Therefore, each node has two labels, a flag and a degree, that denote whether the node is dead or alive and the number of edges it has to other alive nodes in the graph. A work-list maintains the currently alive nodes. The algorithm is executed in rounds. In each round, every node in the work-list marks itself as dead if its degree is less than k or adds itself to the

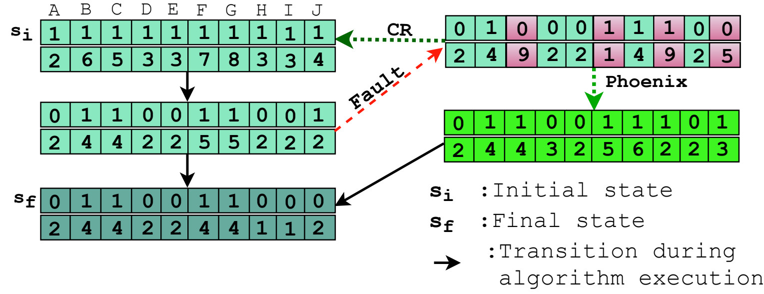


Figure 5.6: State of the graph in Figure 5.1(a), treated as undirected, during the execution of data-driven k-core (k = 4).

work-list for the next round otherwise. When a node is marked as dead, it decrements the degrees of its immediate neighbors. The algorithm terminates (*global_termination*) when no node is marked as dead in a round on any host.

To present the fault recovery strategy for this algorithm, we first note that the state is valid if and only if (i) the degree of each node is equal to the number of the edges it has to currently alive nodes and (ii) all the currently alive nodes are in the work-list. During algorithm execution, a node's degree is updated only when its neighbor changes (its flag) from alive to dead. Hence, when a node is lost, re-initializing its flag and degree does not restore the state to a valid state: the degree of all nodes needs to be recalculated based on currently alive nodes. To do so, new functions are required to reinitialize the degree of a healthy node and recompute the degree of a node using only the current flags (alive or dead) of its neighbors (in Algorithm 5, `ReInitW` and `ReComputeW`).

Figure 5.6 shows state transitions after each round of execution for the graph in Figure 5.1(a). Consider detection of a fault after two rounds. CR

restores the state to that at the end of the first round. In contrast, Phoenix resets both the flag and the degree of proxies on failed hosts to the initial value and resets only the degree of the proxies on healthy hosts to the initial value. All proxies are added to the work-list in both. `SyncW` is called to recover the flags of nodes on failed hosts that have proxies on healthy hosts. After that, all the dead nodes decrement the degree of their outgoing-neighbors and all the alive nodes are added to the work-list. The subsequent `SyncW` will synchronize the degrees of the proxies and restore the state to a valid state. Algorithm execution then converges in a round.

For all globally-correcting algorithms, a node's state is updated using the change of its neighbors' state. In kcore, degree is updated using the change in flag, while in residual-based pagerank (pr), residual is updated using the change in rank. Due to this, re-initializing the labels of failed proxies is insufficient. Some labels of healthy proxies must also be reinitialized. In kcore and pr, only degree and residual are reinitialized, respectively, which ensures that the progress of flag and rank, respectively, on healthy proxies is not lost. The new state must be recomputed using only the current state. In kcore and pr, degree and residual must be recomputed using only the current flag and rank, respectively. Given re-initialization and re-computation functions, the Phoenix API, defined in Algorithm 6, supports such confined recovery.

5.5.5 Phoenix API

The Phoenix API is specific to the class of algorithm, as defined in Algorithms 2, 4, and 6. For self-stabilizing algorithms, the API takes the initialization function as input and updates the state. For locally-correcting algorithms, the API takes the initialization function as input and updates both the state and the work-list. For globally-correcting algorithms, the API updates the state and the work-list by taking functions for initialization, re-initialization, and re-computation as input.

Instrumenting self-stabilizing and locally-correcting algorithms to enable Phoenix is straight-forward because the initialization function required by the API would be used in fault-free execution regardless. On the other hand, for globally-correcting algorithms, the programmer must write new re-initialization and re-computation functions to use the Phoenix API. These functions can be written by considering algorithm execution in shared-memory where node labels are corrupted, which is much simpler than considering algorithm execution in distributed-memory with synchronization of proxies. This typically involves writing a naive topology-driven algorithm instead of the data-driven, work-efficient algorithm used by default. This can be learned from the pattern in Algorithm-5.

While we described Phoenix using BSP-style rounds, the recovery mechanisms of Phoenix do not assume that the execution preceding or following the recovery is BSP-style. Phoenix recovery itself is BSP-style and involves at least one computation and communication round. Phoenix restores the state

to a valid state, regardless of whether the prior updates to the state were bulk-synchronous or asynchronous.

The Phoenix API replaces the checkpoint and restore functions in Checkpoint-Restart (CR) systems, and it can be incorporated into existing synchronous or asynchronous distributed graph analytics programming models or runtimes. Phoenix can also be combined with existing checkpointing techniques that take globally consistent or locally consistent snapshots [106, 171]. In this case, the failed hosts initialize their labels from the saved checkpoint instead of calling the initialization function, thereby leading to faster recovery of nodes for which no proxy exists on a healthy host.

5.6 Experimental Evaluation

D-Galois [51, 118] is the state-of-the-art distributed graph analytics system, but it does not support fault tolerance. The Phoenix fault tolerance approach presented in this work was implemented in D-Galois (adding ~ 500 lines of code); for brevity, the resulting system is called Phoenix too. We also implemented a checkpoint-restart technique in D-Galois, and this is termed CR. We compare these with GraphX [181] and Gemini [195], which are fault tolerant and fault intolerant distributed graph analytics systems respectively. Section 5.6.1 describes the experimental setup, including implementation details of Phoenix and CR. Sections 5.6.2 and 5.6.3 present the results in the presence and absence of faults respectively.

	Stampede	Wrangler
NIC	Omni-path	Infiniband
Machine	Intel Xeon Phi KNL	Intel Xeon Haswell
No. of hosts	128	32
Threads per host	272	48
Memory	96GB DDR4	128GB DDR4
Compiler	g++ 7.1	g++ 4.9.3

Table 5.2: Configuration of clusters.

	amazon	twitter50	rmat28	kron30	clueweb12	wdc12
$\|V\|$	31M	51M	268M	1,073M	978M	3,563M
$\|E\|$	82.5M	1,963M	4,295M	10,791M	42,574M	128,736M
$\|E\|/\|V\|$	2.7	38	16	16	44	36
max D_{out}	44,557	779,958	4M	3.2M	7,447	55,931
max D_{in}	25,366	3.5M	0.3M	3.2M	75M	95M
Size on Disk (GB)	1.2	16	35	136	325	986

Table 5.3: Inputs and their key properties.

5.6.1 Experimental Setup

Experiments were conducted on the Stampede [155] and Wrangler clusters at the Texas Advanced Computing Center [4]. The configurations used are listed in Table 5.2. We used Wrangler to compare Phoenix with D-Galois, Gemini, and GraphX (GraphX cannot be installed on Stampede). All other experiments were conducted on 128 KNL hosts of Stampede.

Table 5.3 specifies the input graphs: wdc12 [110, 111] and clueweb12 [19, 20, 134] are the largest publicly available web-crawls; kron30 [102] (Kronecker), and rmat28 [36] (RMat) are randomized synthetic scale-free graphs (we used weights of 0.57, 0.19, 0.19, and 0.05, as suggested by graph500 [1]); twitter50 [21] is a social network graph; amazon [74] is the largest publicly avail-

Table 5.4: Fault-free execution of Phoenix and CR (R stands for the number of BSP-style rounds).

App	Input	R	Total Time (s)			Execution Time (s)		
			Phoenix	**CR-50**	**CR-500**	**Phoenix**	**CR-50**	**CR-500**
cc	**clueweb12**	24	73.1	72.8	78.5	12.6	14.8	17.4
	wdc12	401	303.8	356.5	306.1	90.5	146.0	95.5
kcore	**clueweb12**	680	254.5	309.3	266.9	166.4	221.7	177.6
	wdc12	270	563.6	625.7	573.0	269.5	334.2	279.5
pr	**clueweb12**	570	290.8	326.8	303.0	243.7	280.6	249.8
	wdc12	747	871.4	1012.8	893.2	732.2	875.1	748.4
sssp	**clueweb12**	200	85.1	87.6	85.3	26.0	32.8	30.5
	wdc12	3779	777.8	1183.7	822.2	620.1	1020.2	659.2
cf	**amazon**	908	266.1	342.2	273.3	254.1	330.5	262.3

able bipartite graph. Smaller inputs are used for comparison with GraphX because it exhausts memory quickly.

Our evaluation uses 5 benchmarks: connected components (cc), k-core decomposition (kcore), pagerank (pr), single-source shortest path (sssp), and collaborative filtering using stochastic gradient descent (cf). In D-Galois (consequently, Phoenix and CR), cf is a self-stabilizing algorithm, cc and sssp are locally-correcting algorithms, and kcore and pr are globally-correcting algorithms. We used GraphX and Gemini implementations of the same benchmarks (they use self-stabilizing algorithm for pr); they do not have kcore or cf. GraphX does not use edge-weights in its sssp, so Gemini and Phoenix also do not use it when compared with it. We present cf results only with the amazon graph because it requires a bipartite graph. The tolerance used for cf is 10^{-9}. The source node for sssp is the maximum out-degree node. The tolerance for pr is 10^{-7} for kron30, 10^{-4} for clueweb12, and 10^{-3} for wdc12. The k in kcore is 100. *All benchmarks are run until convergence.* We present the mean of 3

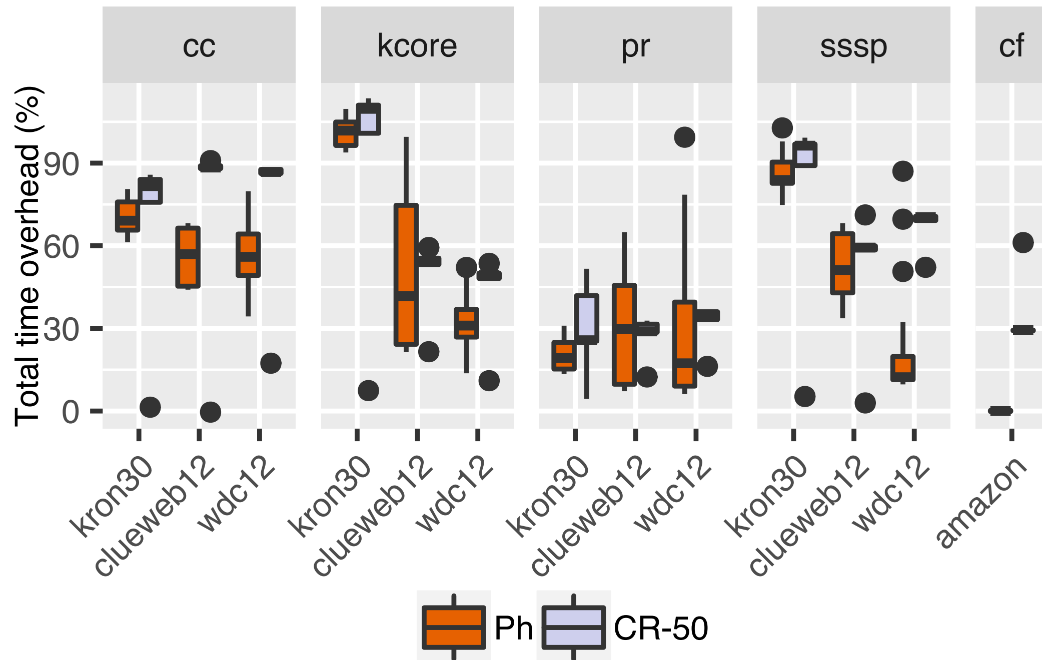


Figure 5.7: Overheads in total time (%) of different fault scenarios with Phoenix and CR over fault-free execution.

runs.

We implemented Phoenix recovery for each benchmark depending on the class of its algorithm. Most of the effort was spent for kcore and pr, which are globally-correcting algorithms. They took an estimated day's worth of programming with $\sim$ 150 lines changed or added (original code was $\sim$ 300 lines). The rest of the benchmarks were self-stabilizing or locally-correcting algorithms, so they took little time and needed only $\sim$ 30 lines of changes or additions.

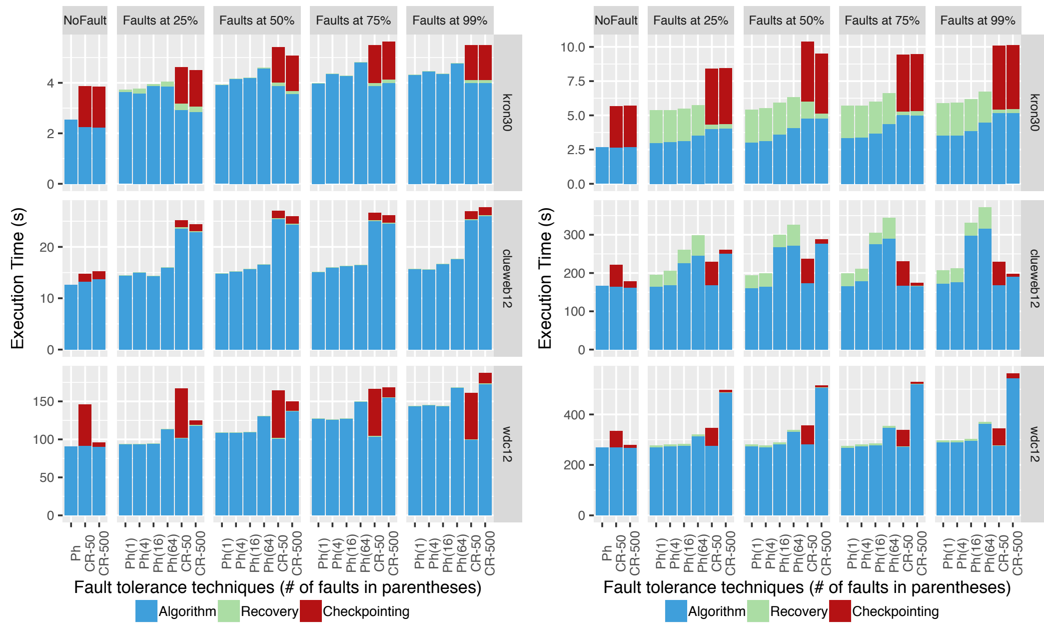
CR only checkpoints the graph node labels and not the graph topology (which is read-only). Since D-Galois is bulk-synchronous parallel (BSP),

CR takes a globally consistent snapshot soon after bulk-synchronization and checkpoints it to the Lustre network filesystem [2] using stripe count 22 and stripe size 2 MB (Asynchronous checkpointing techniques like CoRAL [171] that do not take a globally consistent snapshot cannot be used because it is not applicable for all benchmarks we evaluate). The periodicity of checkpointing can be specified at runtime. We evaluated checkpointing after every 5, 50, and 500 rounds of BSP-style execution, and these are called CR-5, CR-50, and CR-500 respectively.

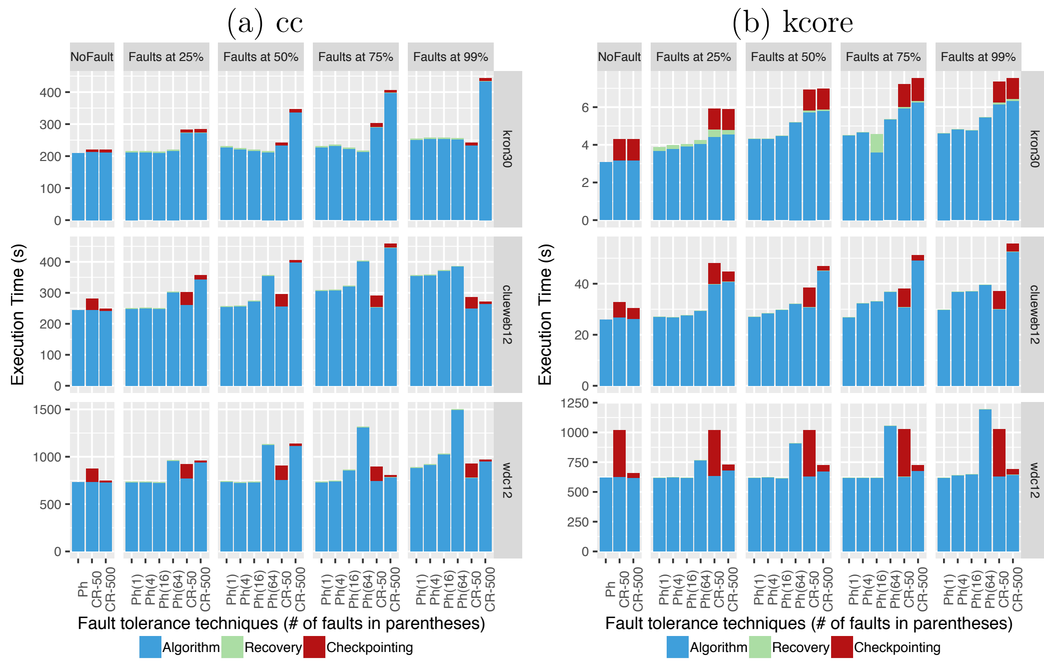
5.6.2 Fault-free performance

Table 5.1 compares the performance of GraphX, Gemini, D-Galois, and Phoenix on Wrangler in the absence of faults. All systems read the graph from a single file on disk and partition it among hosts. The total time includes the time to load and partition the graph. *Phoenix is on average* $\sim 24\times$ *and* $\sim 4\times$ *faster than GraphX and Gemini, respectively. Using the Phoenix substrate to make D-Galois resilient adds no overhead during fault-free execution.* GraphX is more than an order of magnitude slower than Phoenix during fault-free execution, and Gemini and D-Galois do not tolerate faults, so we do not evaluate them with different fault scenarios.

For the rest of our experiments, we assume that the graph is already partitioned, and we load graph partitions directly from disk. Table 5.4 compares the fault-free performance of Phoenix and CR on Stampede (D-Galois is omitted as it is identical to Phoenix). We present the total time and execution



(a) cc (b) kcore



(c) pr (c) sssp

Figure 5.8: Execution time (s) of Phoenix (Ph) and CR with different fault scenarios.

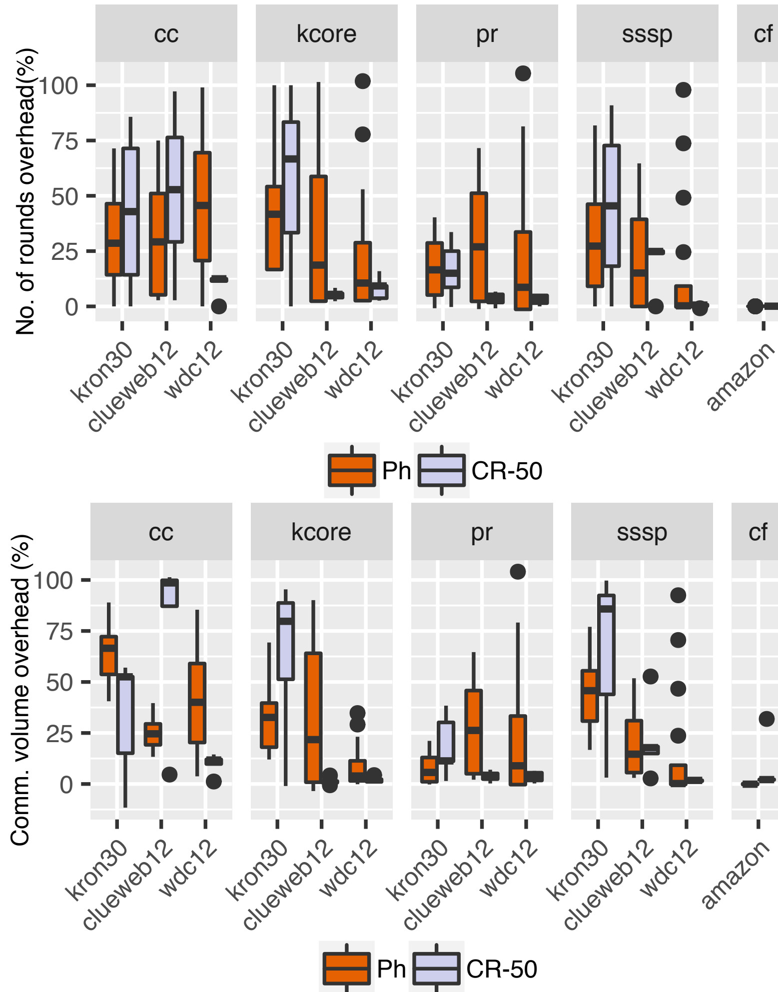


Figure 5.9: Overheads of different fault scenarios with Phoenix and CR over fault-free execution.

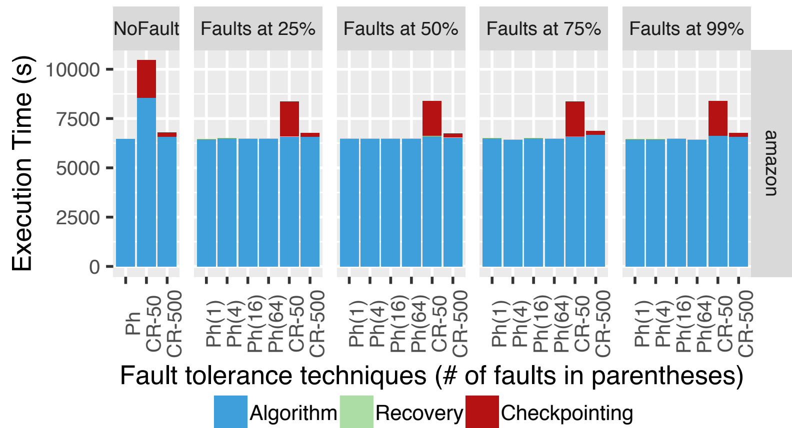


Figure 5.10: Execution time (s) of Phoenix (Ph) and CR with different fault scenarios for cf.

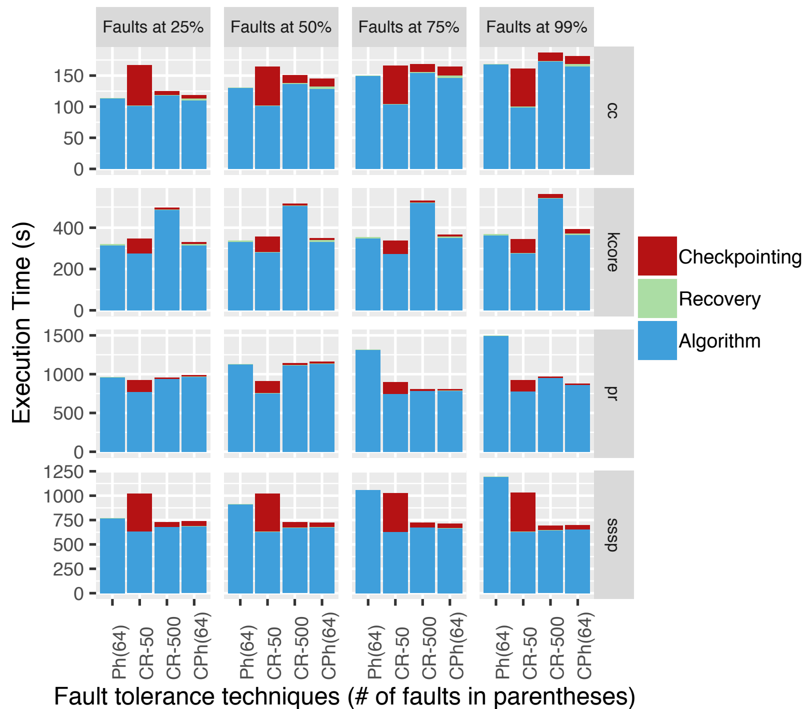


Figure 5.11: Execution time (s) of Phoenix (Ph), CR, and Checkpointing with Phoenix (CPh): 64 faults for wdc12.

time separately. Execution time includes the algorithm time and checkpointing time, while total time includes graph loading time and execution time. Phoenix has 0% overhead in the absence of faults. We omit CR-5 from the table since its overhead is too high. *CR-5, CR-50, CR-500 have an execution time overhead of 542%, 31%, and 8%, respectively, on the average, over Phoenix in the absence of faults.*

5.6.3 Overhead of fault tolerance

To evaluate the behavior of Phoenix and CR when fail-stop faults occur, we simulate a fault by clearing all in-memory data structures on the failed hosts and running recovery techniques on the failed hosts (rebirth-based recovery). We simulate various fault scenarios by varying the number of failed hosts (1, 4, 16, 64 hosts) as well as by causing the hosts to fail at different points in the algorithm execution (failing after executing 25%, 50%, 75%, 99% of rounds). *These fault scenarios are used for all results presented in this section, including Figures 5.7, 5.8, 5.9, 5.10, and 5.11.* The number of failed hosts does not impact CR because all hosts rollback to the last checkpoint (the graph partition re-loading time does not vary much because we are limited by the host's bandwidth, not the network's bandwidth). Note that these scenarios are designed to test the best and worst fault scenarios for Phoenix, and we do not choose the point of failure to test the best or worst fault scenarios for checkpointing.

For each benchmark and input, Figure 5.7 shows the total time overhead

of faults for Phoenix and CR-50 over fault-free execution of Phoenix using a box-plot[4] to summarize all fault scenarios. We omit CR-5 and CR-500 because the overheads are more than 100% in several cases. Phoenix has lower overhead than CR-50 in most cases. For all benchmarks, the overhead reduces as the size of the graph increases. Smaller graphs have very little computation, so almost all of the recovery overhead is from re-loading the partitions from disk on the failed hosts. Except for the smaller kron30, Phoenix has at most 50% overhead when faults occurs in most cases.

We now analyze the overhead of faults with Phoenix and CR in more detail by comparing execution time, which excludes initial loading of graph partitions as well as re-loading of graph partitions on failed hosts. Figures 5.8 and 5.10 compare the execution time of Phoenix and CR in different fault scenarios, including no faults. Execution time is divided into algorithm, recovery, and checkpointing time. Recovery time in Phoenix and CR is the time to restore the state to a valid state and globally consistent state, respectively. We omit CR-5 in the figure because its overhead is too high. We observe that for realistic fault scenarios [147] such as up to 16 hosts failing, Phoenix outperforms CR-50 and CR-500, except for kcore on clueweb12. *When 4 hosts fail, the mean execution time overhead over fault-free execution of Phoenix, CR-50, and CR-500 is ∼14%, ∼48.5%, and ∼59%, respectively.* CR-500 is worse than CR-50 in many cases because it loses more progress when faults occur due to

[4] The box represents the range of 50% of the values, the line dividing the box is the median of those values, and the circles are outliers.

less frequent checkpointing. For Phoenix, as we increase the number of failed hosts, the execution overhead increases. The mean execution time overhead for 16 failed hosts is ∼20.8%, and it increases to ∼44% when 64 hosts fail. In contrast, the point of failure does not increase the overhead of CR by much in most cases. The takeaway is that even in the worst case scenario for Phoenix (64 hosts fail after 99% of rounds), the performance of Phoenix is comparable or better than CR.

When some state is lost due to faults, Phoenix and CR-50 restore the state to a valid and globally consistent state, respectively. However, the algorithm may need to execute more computation and communication to recover the lost state. For Phoenix and CR-50, Figure 5.9 shows the % increase in rounds and communication volume due to failures over fault-free execution of Phoenix (box-plot summarizes all fault scenarios). The overhead is under 100% for Phoenix and CR-50; therefore, both schemes of fault tolerance are better than simply re-executing applications in case of failure. In most cases, Phoenix has lower overhead than CR-50. Furthermore, the overhead of Phoenix is less than 50% in almost all cases.

Combining Phoenix with checkpointing: As observed in our analysis of CR-50 and CR-500, reducing the frequency of checkpointing reduces overhead in fault-free execution, but it can lead to high overheads in case of failure as more progress can be lost. On the other hand, Phoenix has no overhead in fault-free execution, but in worst case scenarios (such as 64 hosts crashing after

99% of execution), it can have considerable overhead. To overcome this, we can combine Phoenix with checkpointing (CPh) so that we take checkpoints less frequently (every 500 rounds), but in case of failures, crashed hosts can rollback to the last saved checkpoint, and all hosts can use Phoenix to help crashed hosts recover faster. Figure 5.11 shows the evaluation of CPh in case of 64 hosts crashing at different points of execution for all benchmarks on wdc12. CPh is similar to or faster than CR-500 in all cases, as expected.

5.7 Related Work

System-level checkpointing: Many systems [9, 44, 60, 69, 95, 104, 108, 121, 137, 143] implement checkpointing or message logging to rollback and recover from faults [59]. Although these system-level mechanisms are transparent to the programmer, exploiting application properties like Phoenix does can reduce the execution time with or without faults.

Application-level checkpointing: Instead of checkpointing the entire state or memory footprint of the application like in system-level checkpoint-restart approaches, many checkpointing approaches [115, 159, 191] allow the programmer to instrument their code to checkpoint only the live application state; Bronevetsky et al. [29–31] automate this using a compiler. Some of these approaches checkpoint to memory [159, 191] or use a combination of memory, local disk, and network filesystem [115]. In Section 5.6, we showed that in the presence of faults, Phoenix generally outperforms even a checkpoint-restart

(CR) approach that checkpoints only node labels (and not the graph topology) to the network filesystem. Although CR can be further improved by using in-memory or multi-level checkpointing, there will always be overhead even in fault-free execution, unlike in Phoenix.

Fault tolerant data-parallel systems: Some data-parallel systems [55, 82, 132, 186] save sufficient information transparently to re-execute computation and restore lost data when faults occur. Schelter et al. [146] allow users to specify functions that can recover state when faults occur, but their technique is applicable only to self-stabilizing and locally-correcting algorithms. Phoenix, in contrast, does not require the user to define *new* functions for these classes of algorithms, and it supports globally-correcting algorithms with user-defined functions. Moreover, Phoenix does not store any additional state information in memory or stable storage.

Fault tolerant graph-analytical systems: Many systems for distributed graph analytics [68, 70, 103, 106, 132, 133, 136, 148, 149, 171, 172, 181] support fault tolerance transparently. Greft [133] is the only one that tolerates Byzantine (data corruption) faults; the rest tolerate only fail-stop faults. Section 5.3.1 contrasts Phoenix with other fail-stop fault tolerant systems in detail.

Fault tolerant algorithms: Algorithm-Based Fault Tolerance (ABFT) approaches [24, 41, 42, 53, 81, 173, 179, 184] have modified several computational

science algorithms to tolerate faults without checkpointing. In a similar vein, many iterative solvers [77, 123, 145] have used self-stabilization [57] to tolerate failures. Sao et al. [144] build on this to design and implement a self-correcting topology-driven connected components algorithm. Some of these algorithms detect and recover from data corruptions (transient soft faults) in shared-memory systems. Phoenix generalizes the concept of self-stabilizing and self-correcting algorithms as explained in Section 5.4. In addition, Phoenix provides a simple API to implementing such algorithms on distributed-memory as explained in Section 5.5.

Chapter 6

Distributed Training of Embeddings using Graph Analytics[1]

6.1 Motivation

Many applications today, such as natural language processing, network analysis, and code analysis, rely on semantically embedding objects into low-dimensional fixed-length vectors. Such embeddings naturally provide a way to perform useful downstream tasks, such as identifying relations among objects or predicting objects for a given context, etc. Word2Vec [112, 113] is a popular algorithm for learning word embeddings. Follow-on work extends the ideas of mapping entities to embeddings to biological sequences[93], program code[11], and online social networks[127], and collectively these machine learning algorithms are referred to as Any2Vec.

Unfortunately, the training necessary for accurate embeddings is usually computationally intensive and requires processing large amounts of data. For example, some of the largest datasets used in this work take days to

[1]This work was originally published in the proceedings of CS arXiv, 2019 [67]. The main idea of formulating word3vec training problem as a graph application was conceived by the first author. The first author was also responsible implementation and experimentation while the co-authors helped with the presentation.

train sequentially on a single machine. Furthermore, distributing this training is challenging. Most embedding training uses stochastic gradient descent (SGD) [25, 26], an "inherently" sequential algorithm where at each step, the processing of the current example depends on the parameters learned from the previous examples. Prior approaches to parallelizing SGD do not honor these dependencies and thus potentially suffer poor convergence.

6.2 Contributions

This work presents a distributed training framework for a class of applications that use *Skip-gram-like* models, like the one used in Word2Vec [112], to generate embeddings. We call this class Any2Vec and includes, in addition to Word2Vec [112], DeepWalk [128] and Node2Vec [72] among others. Applications in this class maintain a large embedding matrix, where each row corresponds to the embedding for each object. Given sequences of objects (text segments for Word2Vec and graph paths in DeepWalk), the training involves looking up the embedding matrix for the objects in the sequence and updating them through stochastic gradient descent (SGD). The details of the how the sequences are generated and the cost functions used to update the embeddings varies with the application.

The key challenge in distributing Any2Vec training is that SGD is *inherently* sequential. Two approaches for parallelizing SGD are asynchronous SGD, where multiple nodes racily update [138] a model that may be housed in a global parameter server [54], or synchronous SGD, where nodes bulk-

synchronously combine individual gradients in a mini-batch update before updating the model [8]. It is well known that the staleness of updates affects the scalability of the former, while the increase in mini-batch size affects the scalability of the latter. This is substantiated in our evaluation.

To improve the scalability over prior methods, this work introduces ***GraphAny2Vec***, a distributed machine learning framework for Any2Vec. We first demonstrate that the Any2Vec class of machine learning algorithms can be formulated as a graph application and leverage the ease of programming and scalability of the state-of-the-art distributed graph analytics frameworks, such as D-Galois [51] and Gemini [195]. To support this new application, we extend D-Galois to support dynamic graph generation and re-partitioning, and implement communication optimizations for reducing the communication volume, the main bottleneck for these applications at scale. Finally, we introduce a novel way to combine gradients during distributed training to prevent accuracy loss when scaling. Rather than simply averaging the gradients, as in a synchronous mini-batch SGD, our Gradient Combiner (GC) performs a weighted combination on gradients based on whether they are parallel or orthogonal to each other.

We evaluate two applications: Word2Vec [112] and Vertex2Vec [128], in our GraphAny2Vec framework on a cluster of up to 32 machines with 3 different datasets each. We compare GraphAny2Vec training time and accuracy with the state-of-the-art shared-memory implementations (original C implementation [113] and Gensim [139] for Word2Vec and DeepWalk [128]

for Vertex2Vec) as well as with the state-of-the-art distributed parameter-server Word2Vec implementation in Microsoft's Distributed Machine Learning Toolkit (DMTK) [160]. We show that compared to shared-memory implementations, GraphAny2Vec can reduce the training time for Word2Vec from 21 hours to less than 2 hours on our largest dataset of Wikipedia articles while matching the SGD accuracy of shared-memory implementations, and gives a geo-mean speedup of 12$\times$ and 5$\times$ for Word2Vec and Vertex2Vec respectively. On 32 hosts, GraphAny2Vec is on average 2$\times$ faster than DMTK. We also show the superiority of our *Gradient Combiner* (GC) independent of GraphAny2Vec by incorporating it in DMTK, which raises its accuracy by $> 30\%$ so that it matches its own shared-memory implementation.

6.3 Background

In this section, we first briefly describe how stochastic gradient descent is used to train machine learning models (Section 6.3.1), followed by how Any2Vec models are trained with Word2Vec as an example (Section 6.3.2). We then provide an overview of graph analytics (Section 6.3.3).

6.3.1 Stochastic Gradient Descent

We express the training task of a machine learning model as a set of multivariable loss functions $L_i(w) : \mathbb{R}^n \rightarrow \mathbb{R}$ where w is the model and each L_i corresponds to the training sample i. The output of $L_i(w)$ is a positive value that correlates the prediction of the model w to the label of sample i. Perfect

prediction has a loss of 0. The ultimate goal is to find w that minimizes the loss function across all samples: $\arg\min_w : \sum_i L_i(w)$.

Stochastic Gradient Descent (SGD) [25] is a popular algorithm for machine learning training. The model is initially set to a random guess w_0 and at iteration or sample i,

$$w_i := w_{i-1} - \alpha \cdot \left.\frac{\partial L_i}{\partial w}\right|_{w_{i-1}}$$

where α is the *learning rate* and $\left.\frac{\partial L_i}{\partial w}\right|_{w_{i-1}}$ is the *gradient* of L_i at w_{i-1}. Training is complete when the model reaches a desired loss or evaluation accuracy. An *epoch* of training is the number of updates needed to go through the whole dataset once.

The fact that SGD's update rule for w_i depends on w_{i-1} makes SGD an inherently sequential algorithm. A well-known technique to introduce parallelism is *mini-batch* SGD [27], wherein the gradient is calculated as an average over n training examples, where n is the *mini-batch size*. When n is 1 this is equivalent to normal SGD.

Hogwild! [138] is another well-known SGD parallelization technique, wherein multiple threads compute gradients for different training examples in parallel and update the model in a racy fashion. Surprisingly, this approach works well on a shared-memory system, especially with models where gradients are sparse.

This work uses intuitions based on the **Taylor expansion of SGD** to develop new techniques for parallelizing SGD. Applying the SGD update rule

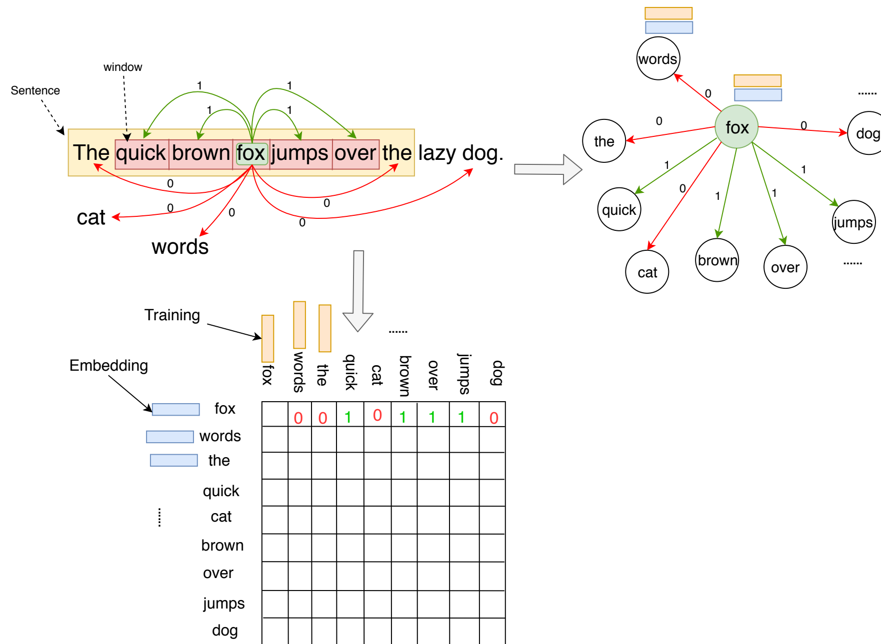


Figure 6.1: Viewing Word2Vec in Skip-gram model as a graph.

to a loss function L_i and expanding with the Taylor approximation gives:

$$\begin{aligned} L_i(w_i) &= L_i(w_{i-1} - \alpha \cdot g_i) \approx L_i(w_{i-1}) - \alpha \cdot g_i^T \cdot \frac{\partial L_i}{\partial w}\bigg|_{w_{i-1}} \\ &= L_i(w_{i-1}) - \alpha \cdot g_i^T \cdot g_i \\ &= L_i(w_{i-1}) - \alpha \cdot \|g_i\|^2 \end{aligned} \tag{6.1}$$

As it is clear from Equation 6.1, moving in the direction of the gradient reduces the loss. Note that the learning rate, α, is a delicate hyper-parameter: a small α decays the loss insignificantly and for large α, the Taylor expansion approximation breaks and the model diverges.

6.3.2 Training Any2Vec Embeddings

An embedding is a mapping from a dataset D to a vector space $\mathbb{R}^N$ such that elements of D that are related are close to each other in the vector space. The length N of the *embedding vectors* is typically much smaller than the dimension of D.

Many models have been proposed for learning word embeddings [112, 113]. We will focus on the popular Skip-gram model together with the negative sampling introduced in [113], and explain it here in a form suitable for a graph analytic understanding.

Skip-gram uses a training task where it predicts if a target word w_O appears in the context of a center word w_I. Figure 6.1 (left) illustrates this for an example sentence, with "fox" as the center word. The context is then defined as the words inside a window of size c (a hyper-parameter) centered on "fox". In Figure 6.1 these *positive* samples are shown in green and have a label of 1. For each positive sample Skip-gram picks k (a hyper-parameter) random words as *negative* samples and gives them a label of 0.

The Skip-gram model consists of two vectors of size N for each word w in the vocabulary: an *embedding* vector e_w and a *training* vector t_w. For a pair of words the model predicts the label with $\sigma(e_{w_I}^T \cdot t_{w_O})$, which should be close to 1 for related words and close to 0 otherwise. The loss term for a sample is then $-\log(1 - |y - \sigma(e_{w_I}^T \cdot t_{w_O})|)$, where y is the true label of the sample.

We base the work in this chapter on Google's Word2Vec tool[2], which uses the Hogwild! parallelization technique. Each thread is given a subset of the corpus and goes through the words in it sequentially (skipping some due to sub-sampling frequent words as described in [113]). For each pair of a central word and a target word in its context Word2Vec calculates a gradient using a sum of the loss term for the positive sample itself and loss terms for k negative samples. This gradient is then applied to the model shared by all threads in a racy manner and the thread continues onto the next pair of words.

6.3.3 Graph Analytics

In typical graph analytics applications, each node has one or more labels, which are updated during algorithm execution until a global quiescence condition is reached. The labels are updated by iteratively applying a computation rule, known as an *operator*, to the nodes or edges in the graph. The order in which the operator is applied to the nodes or edges is known as the *schedule*. A *node operator* takes a node n and updates labels on n or its neighbors, whereas an *edge operator* takes an edge e and updates labels on source and destination of e.

To execute graph applications in distributed-memory, the edges are first partitioned [75] among the hosts and for each edge on a host, proxies are created for its endpoints. As a consequence of this design, a node might have proxies (or replicas) on many hosts. One of these is chosen as the *master*

[2] https://code.google.com/archive/p/word2vec/

proxy to hold the canonical value of the node. The others are known as *mirror* proxies. Several heuristics exist for partitioning edges and choosing *master* proxies [65].

Most distributed graph analytics systems [51, 70, 195] use bulk-synchronous parallel (BSP) execution. Execution is done in rounds of computation followed by bulk-synchronous communication. In the computation phase, every host applies the *operator* inside its own partition and updates the labels of the local proxies. Thus, different proxies of the same node might have different values. Every host then participates in a global communication phase to synchronize the labels of all proxies. Different proxies of the same node are reconciled by applying a *reduction* operator, which depends on the algorithm being executed.

6.4 Distributed Any2Vec

In this section, we first describe the formulation of Any2Vec as a graph application and provide an overview of our distributed GraphAny2Vec (Section 6.4.1). We then describe the different phases in our approach such as dynamic graph generation and partitioning (Section 6.4.2), model synchronization (Section 6.4.3), and communication optimizations (Section 6.4.4).

6.4.1 Overview of Distributed GraphAny2Vec

We formulate Any2Vec as a graph problem and call it GraphAny2Vec. Each element in the dataset corresponds to a node in a graph, and the positive

Procedure `GraphAny2Vec(`*Corpus C, Num. of epochs R, Num. of sync round S, Learning rate α*`):`

Let h be the host ID
Stream C from disk to build set of vertices V
Read partition h of C that forms the work-list of vertices W
Build graph G from V
for *epoch r from 1 to R* **do**
  for *sync round s from 1 to S* **do**
    Let WL_s be partition s of WL Build graph $G = (V, E)$ where E are samples in W_s Compute(G, W_s, α) ▷ Updates G and decays α
    Synchronize(G) ▷ Updates G
  end
end
Return

algorithm 7: Execution on each distributed host of GraphAny2Vec.

and negative samples correspond to edges in the graph with weights 1 and 0 respectively. Figure 6.1 (right) illustrates this for Word2Vec. Training the Skip-gram model is now a graph analytics application. Each node has two labels — e and t — for embedding and training vectors, respectively, of size N. The model corresponds to these labels for all nodes. These labels are initialized randomly and updated during training by applying an *edge operator*, that takes the source *src* and destination *dst* of an edge with weight w, computes $\sigma(e^T_{\texttt{src}} \cdot t_{\texttt{dst}})$ to predict the relation between the two nodes, and then applies the SGD update rule to $e_{\texttt{src}}$ and $t_{\texttt{dst}}$ so as to minimize the loss function $-\log(1 - |w - \sigma(e^T_{\texttt{src}} \cdot t_{\texttt{dst}})|)$. The operator is applied to all edges once in each epoch.

Algorithm 7 gives a brief overview of our distributed GraphAny2Vec execution. The first step is to construct the set of vertices V (unique words in case of Word2Vec) by making a pass over the training data corpus C on each

host in parallel. As C may not fit in the memory of a single host, we stream it from disk to construct V. The corpus C is then partitioned (logically) into roughly equal contiguous chunks among hosts. All hosts read their own partition of C in parallel. The list of elements in a given host's partition of C constitutes the work-list[3] W that the host is responsible for computing Any2Vec on. We introduce a new parameter for controlling the number of synchronization rounds within an epoch. In each epoch on each host, W is partitioned into roughly equal contiguous chunks among the rounds. In each round s, positive and negative samples from partition s of W are used to construct the graph. The Any2Vec *operator* is then applied to all edges in the graph. The operator updates the vertex labels directly and decays the learning rate continuously, as in shared-memory implementation of Any2Vec applications. Then, all hosts participate in a bulk-synchronous communication to synchronize the vertex labels.

We implement GraphAny2Vec in D-Galois [51], the state-of-the-art distributed graph analytics framework, which consists of the Galois [118] multi-threaded library for computation and the Gluon [51] communication substrate for synchronization. Galois provides efficient, concurrent data structures like graphs, work-lists, dynamic bit-vectors, etc., which makes it quite straightforward to implement GraphAny2Vec. Gluon incorporates communication optimizations that enable it to scale to a large number of hosts. However, D-

[3]The work-list W does not change across epochs, so we construct it once and reuse it for all epochs and synchronization rounds. However, if it does not fit in memory, partition s of W can be constructed from the corpus in each synchronization round.

Galois only works with static graphs (nodes and edges must not change during algorithm execution), whereas, for Any2Vec applications, edges are sampled randomly and generated. We adapted D-Galois to handle dynamic graph generation efficiently during computation and communication, as explained in Sections 6.4.2 and 6.4.4 respectively. Our techniques can be used to modify other distributed graph analytics frameworks and implement GraphAny2Vec in them.

6.4.2 Graph Generation and Partitioning

As explained in Section 6.3.2, the Skip-gram model generates positive and negative samples using randomization. Consequently, the samples or edges generated for the same element or node in the corpus in different epochs may be different. As the same edge may not be generated again, one way to abstract this is to consider that the edges are being streamed and each edge is processed only once, even across epochs. Due to this, the graph needs to be constructed in each synchronization round, as shown in Algorithm 7.

The graph can be explicitly constructed in each round. However, this may add unnecessary overheads as each edge is processed only once before the graph is destroyed. More importantly, this does not distinguish between edges (samples) from different occurrences of the same node (element) in the corpus. Consequently, the relative ordering of the edges from different nodes is not preserved. We observed that the accuracy of the model is highly sensitive to the order in which the edges are processed because the learning rate decays

after each occurrence of the node is processed. Hence, the key to our graph formulation is that on each host, *the schedule of applying operators on edges in GraphAny2Vec must match the order in which samples would be processed in Any2Vec.* Note that the work-list W preserves the ordering of element occurrences in the corpus. Thus, GraphAny2Vec generates or streams edges on-the-fly using partition s of W in round s, instead of constructing the graph.

Each host generates edges for its own partition of the graph in each synchronization round. In other words, the graph is re-partitioned in every round. By design, each edge is assigned to a unique host. As mentioned in Section 6.3.3, node proxies are created for the endpoints of edges on a host. The *master* proxy for each node can be chosen from among its proxies thus created, but this would incur overheads in every round. We instead (logically) partition the nodes once into roughly equal contiguous chunks among the hosts and each host creates master proxies for the nodes in its partition. Proxies for other nodes on the host would be *mirror* proxies. Each mirror knows the host that has its master using the partitioning of nodes. Each master also needs to know the hosts with its mirrors. We provide two ways to do this: RepModel and PullModel.

In **RepModel**, each host has proxies for all nodes, so the entire model is replicated on each host. Thus, each host statically knows that every other host has mirror proxies for the masters on it. This allows GraphAny2Vec to assume that an edge between any two nodes can be generated on any host. In **PullModel**, each hosts makes an inspection pass over W before computation

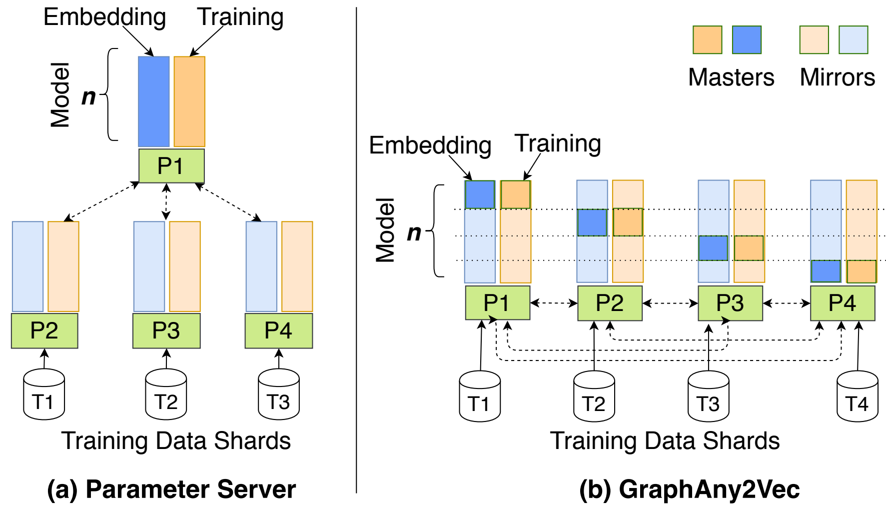


Figure 6.2: Synchronization of the Skip-gram Model.

in each round to generate edges and track the nodes that would be accessed during computation. Mirror proxies are then created for the nodes tracked. For each mirror proxy created, the host communicates to the host that has its master (bulk-synchronization).

RepModel requires the entire model to fit in the memory of a host, while PullModel enables handling larger models. On the other hand, PullModel incurs overhead for determining masters and mirrors in each round, whereas RepModel does not. Nonetheless, RepModel and PullModel require different communication to synchronize the model, so we evaluate which of them performs better in Section 6.7.

6.4.3 Model Synchronization

Prior work for distributed Word2Vec such as Microsoft's Distributed Machine Learning Toolkit (DMTK) [160] (and other machine learning algo-

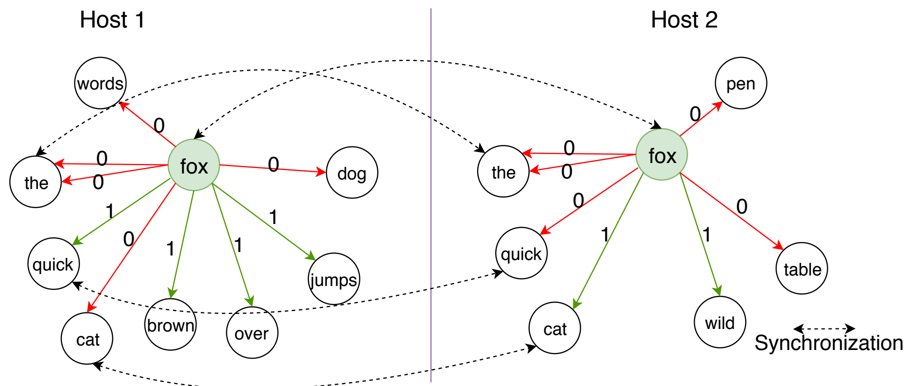


Figure 6.3: Synchronization of proxies in graph partitions.

rithms) use a parameter server to synchronize the model, as illustrated in Figure 6.2(a). One of the hosts (say $P1$) is chosen as the parameter server. At the beginning of a round (or a mini-batch), every host receives the updated model from the parameter server. The host then computes that round and sends the model updates to the parameter server. GraphAny2Vec uses a different synchronization model based on D-Galois [51], as illustrated in Figure 6.2(b). Abstractly, this can be viewed as a generalization of the parameter server model where each host acts as a parameter server for a partition of the model. In Figure 6.2(b), $P1$ has the master proxies for the first contiguous chunk or partition of the nodes, $P2$ has the master proxies for the second partition of the nodes, and so on. During synchronization in D-Galois, the mirror proxies send their updated value to the host containing the master, which reduces it and broadcasts it to the hosts containing the mirrors.

Figure 6.3 shows an example where proxies on two hosts need to be synchronized after computation. Consider the word "*fox*" that is present on both hosts. It may have different values for the embeddings on both hosts after

computation. The reduction operator determines how to synchronize these values and this is a parameter to synchronization in D-Galois. As described in Section 6.5, averaging or adding the two values may lead to slower convergence, so we introduce a novel way to combine them called Gradient Combiner.

6.4.4 Communication Optimizations

RepModel-Naive: During synchronization in RepModel, all mirrors on each host can send their updates to their respective masters and the masters can reduce those values and broadcast it to their mirrors. This is similar to communication for dense matrix codes, so can be mapped quite efficiently to MPI collectives. However, in Any2Vec, not all nodes are updated in every round. Consequently, such naive communication would result in redundant communication during both reduce and broadcast phases.

RepModel-Opt: The advantage of D-Galois is that it allows the user to specify the updated nodes and it would transparently handle the sparse communication that would entail. To do this, we maintain a bit-vector that tracks the nodes that were updated in this round. During synchronization, only the updated mirrors are sent to their masters and the masters broadcast their values to the other hosts only if it was updated on any host in that round. This avoids redundant communication during the reduce phase. However, there is still some redundancy during the broadcast phase because the update sent to a mirror might not be accessed by the mirror in the next round. This information remains unknown in RepModel.

PullModel-Base: In PullModel, mirrors are created after inspection only if one of its labels will be accessed on that host. During synchronization, only the mirrors updated in this round are sent to their masters (similar to RepModel-Opt). However, we wait to broadcast after inspection of the next round when new mirrors are created (re-partitioning). During broadcast, all masters must be broadcast whether updated or not, because previous updates may not have been sent to a host if it did not have a mirror during a previous round. This is essentially pulling the model that will be accessed (like in parameter server). While this avoids sending masters to mirrors that do not access it, it may resend masters that have have been updated.

PullModel-Opt: Recall from Section 6.3.2 that embedding vectors e are accessed only at the source and training vectors t are accessed only at the destination of an edge. If a mirror proxy on a host has only outgoing (or incoming) edges, then it will not access t (or e). This is not exploited in PullModel-Base because masters and mirrors are not label-specific in D-Galois. We modified D-Galois to maintain masters and mirrors specific to each label. We also modified our inspection phase to track sources and destinations separately, and create mirrors for e and t respectively. Due to this, masters will broadcast e and t only to those hosts that access each.

6.5 Gradient Combiner

Section 6.3 discussed how in the mini-batch approach gradients from multiple training examples are computed in parallel and they are reduced to a

single vector by averaging. Although this is a widely-used practice, it does not follow the semantics of the sequential algorithm. Suppose $L_1(w)$ and $L_2(w)$ are two loss functions corresponding to two training examples. Starting from model w_0, sequential SGD calculates $w_1 = w_0 - \alpha \frac{\partial L_1}{\partial w}|_{w_0}$ followed by $w_2 = w_1 - \alpha \frac{\partial L_2}{\partial w}|_{w_1}$ where α is a proper learning rate. With forward substitution, $w_2 = w_0 - \alpha(\frac{\partial L_2}{\partial w}|_{w_1} + \frac{\partial L_1}{\partial w}|_{w_0})$. Alternatively, in a parallel setting, $\frac{\partial L_1}{\partial w}|_{w_0}$ and $\frac{\partial L_2}{\partial w}|_{w_0}$ (note that gradients are both at w_0) are computed and w is updated with $w'_2 = w_0 - \frac{\alpha}{2}(\frac{\partial L_1}{\partial w}|_{w_0} + \frac{\partial L_2}{\partial w}|_{w_0})$. Clearly w'_2 and w_2 are different because of the averaging effect. [71] and [96] have claimed that scaling up the learning rate by the number of parallel processors (or square root of it) closes this gap. However, if $\frac{\partial L_1}{\partial w}|_{w_0}$ and $\frac{\partial L_2}{\partial w}|_{w_0}$ are both in the same direction, scaling up the learning rate might cause divergence as we assumed α was properly set for the sequential algorithm. Our *Gradient Combiner* addresses this problem by adjusting the gradients to each other.

Following the Taylor expansion for $\frac{\partial L_2}{\partial w}|_{w_0+(w_1-w_0)}$, we have:

$$\begin{aligned} \frac{\partial L_2}{\partial w}\Big|_{w_0+(w_1-w_0)} &\approx \frac{\partial L_2}{\partial w}\Big|_{w_0} + \frac{\partial^2 L_2}{(\partial w)^2}\Big|_{w_0} \cdot (w_1 - w_0) \\ &= \frac{\partial L_2}{\partial w}\Big|_{w_0} - \alpha \frac{\partial^2 L_2}{(\partial w)^2}\Big|_{w_0} \cdot \frac{\partial L_1}{\partial w}\Big|_{w_0} \end{aligned} \tag{6.2}$$

where the approximation error is $O(\|w_1 - w_0\|^2)$ which alternatively can be expressed as $\alpha^2 O\Big(\|\frac{\partial L_1}{\partial w}|_{w_0}\|^2 \Big)$. As the learning rate α gets smaller, the error in Formula 6.2 shrinks quadratically. Usually as the training of a Any2Vec model progresses, the learning rate is decayed and as a result this error becomes negligible. For the rest of this section, we denote gradients $g_1 = \frac{\partial L_1}{\partial w}|_{w_0}$,

$g_2 = \frac{\partial L_2}{\partial w}|_{w_0}$, $g'_2 = \frac{\partial L_2}{\partial w}|_{w_1}$ and the Hessian matrix $H_2 = \frac{\partial^2 L_2}{(\partial w)^2}|_{w_0}$. Therefore, Equation 6.2 can be re-written by:

$$g'_2 \approx g_2 - \alpha H_2 \cdot g_1 \tag{6.3}$$

Equation 6.3 lets us compute g'_2, however, computing H_2 is expensive as it is a $n \times n$ matrix where n is the number of parameters in the Any2Vec model. Luckily because Any2Vec has a log-likelihood loss function, H_2 can be expressed by the outer product of the gradient: $\lambda g_2 \cdot g_2^T$ where λ is a scalar which depends on w_0 [192]. The error for this approximation gets smaller as $w_0 \to w^*$ where w^* is the optimal model parameters [73, 192]. By using Equation 6.3 and this approximation, g'_2 can be approximated by:

$$g'_2 \approx g_2 - \alpha \lambda g_2 \cdot g_2^T \cdot g_1 \tag{6.4}$$

Although Formula 6.4 makes calculation of g'_2 feasible, finding the right λ for every iteration of SGD is overwhelmingly difficult and it is yet another hyper-parameter for the user to tune. However, if g_1 was orthogonal to g_2, then g'_2 could have been easily estimated by g_2. This is the intuition behind Gradient Combiner .

Given that g_1 and g_2 are not always orthogonal, we project g_1 on the orthogonal space of g_2 to make g_1^O:

$$g_1^O = g_1 - \frac{g_2^T \cdot g_1}{\|g_2\|^2} g_2 \tag{6.5}$$

g_1^O has three important properties: (1) $g_1^T \cdot g_1^O \geq 0$, (2) $\left\|g_1^O\right\| \leq \|g_1\|$, and (3) $g_2^T \cdot g_1^O = 0$. It is straight forward to check these properties (see Section 6.6

for full proof). Suppose $w_1^O = w_0 - \alpha g_1^O$. Then by using the Taylor expansion, we have:

$$L_1(w_1^O) = L_1(w_0 - \alpha g_1^O) \approx L_1(w_0) - \alpha g_1^T \cdot g_1^O \leq L_1(w_0) \tag{6.6}$$

where the last inequality comes from property (1). This means that moving in the direction of g_1^O decays the loss of L_1. Also, because of property (2) and the fact that the same learning rate as sequential learning rate is used, the approximation in Equation 6.6 has the same or lower error as the one with $L_1(w_1)$ if we had the Taylor expansion for it (refer to Equation 6.1). Property (3) and Equation 6.4 ensure that $\frac{\partial L_2}{\partial w}|_{w_1^O} \approx \frac{\partial L_2}{\partial w}|_{w_0} = g_2$. Let's assume that sequential SGD uses g_1^O to get to $w_1^O = w_0 - \alpha g_1^O$ followed by computing $\frac{\partial L_2}{\partial w}|_{w_1^O}$ which can be approximated by g_2 to get to $w_2^O = w_1^O - \alpha g_2$. By forward substituition:

$$w_2^O = w_0 - \alpha(g_2 + g_1^O) = w_0 - \alpha \left(g_2 + g_1 - \frac{g_2^T \cdot g_1}{\|g_2\|^2} g_2 \right) \tag{6.7}$$

Therefore, the direction Gradient Combiner ($GC(g_1, g_2)$ for short) uses to move is:

$$GC(g_1, g_2) = g_2 + g_1 - \frac{g_2^T \cdot g_1}{\|g_2\|^2} g_2 \tag{6.8}$$

which allows computation of g_1 and g_2 in parallel. Note that our gradient combiner requires slightly more computation than averaging but as we will show in Section 6.7, this overhead is negligible.

The magnitude of $GC(g_1, g_2)$ depends on how parallel or orthogonal g_1 and g_2 are to each other. In the parallel case, g_1^O becomes smaller and

therefore, moving in the direction of g_1^O decays the loss value of L_1 slower than g_1. However, as we will show in Section 6.7, the gradients all start parallel to each other in the begining of the training as they all point in the same general direction and later in the training they become more orthogonal. This means that Gradient Combiner conservatively takes small steps in the begining of the training and larger ones as the training progresses.

Gradient Combiner extends to combining k gradients as well where the gradients are combined in sequentially manner as discussed in Section 6.4.3 ($GC(g_1, \ldots, g_k) = GC(g_k, \ldots, GC(g_3, GC(g_2, g_1)) \ldots)$). Section 6.6.1 proves the convergence of Gradient Combiner in expectation.

The effectiveness of Gradient Combiner depends on the degree to which the gradients are orthogonal. We define

$$O(g_1, g_2) = \frac{\|GC(g_1, g_2)^2\|}{\|g_1\|^2 + \|g_2\|^2} = \frac{\left\|g_1^O + g_2\right\|^2}{\|g_1\|^2 + \|g_2\|^2} \tag{6.9}$$

as a notion for orthogonality of g_1 and g_2. Note that because of property (3) g_1^O and g_2 are orthogonal and $\left\|g_1^O + g_2\right\|^2 = \left\|g_1^O\right\|^2 + \|g_2\|^2$ thanks to Pythagorean theorem. Therefore, $\left\|g_1^O + g_2\right\|^2 = \left\|g_1^O\right\|^2 + \|g_2\|^2 \leq \|g_1\|^2 + \|g_2\|^2$ which concludes that $t \leq 1$ and equality is met only when g_1 and g_2 are orthogonal. On the other hand, if $g_1 = g_2$, g_1^O becomes zero and $O(g_1, g_2) = \frac{1}{2}$. This can be similarly expanded to k gradients $O(g_1, \ldots, g_k) = \frac{GC(g_1, \ldots, g_k)^2}{\sum_i \|g_i\|^2}$. Similarly for k gradients, orthogonality is 1 when they are all orthogonal and $\frac{1}{k}$ when they are all the same.

6.6 Proof of Properties of Gradient Combiner

The three properties of Gradient Combiner are (1) $g_1^T \cdot g_1^O \geq 0$, (2) $\left\|g_1^O\right\| \leq \|g_1\|$, and (3) $g_2^T \cdot g_1^O = 0$ where $g_1^O = g_1 - \frac{g_2^T \cdot g_1}{\|g_2\|^2} g_2$. For the proof, assume that the angle between g_1 and g_2 is θ.

Property (1): $g_1^T \cdot g_1^O \geq 0$.

$$\begin{aligned} g_1^T \cdot g_1^O &= g_1^T \cdot (g_1 - \frac{g_2^T \cdot g_1}{\|g_2\|^2} g_2) = \|g_1\|^2 - \frac{g_2^T \cdot g_1}{\|g_2\|^2} g_1^T \cdot g_2 \\ &= \|g_1\|^2 - \frac{\|g_1\|^2 \|g_2\|^2 \cos^2\theta}{\|g_2\|^2} = \|g_1\|^2 \sin^2\theta \geq 0 \end{aligned} \tag{6.10}$$

Property (2): $\left\|g_1^O\right\| \leq \|g_1\|$.

$$\begin{aligned} \left\|g_1^O\right\|^2 &= \left\|g_1 - \frac{g_2^T \cdot g_1}{\|g_2\|^2} g_2\right\|^2 \\ &= \|g_1\|^2 + \frac{(g_2^T \cdot g_1)^2}{\|g_2\|^4} \|g_2\|^2 - 2\frac{g_2^T \cdot g_1}{\|g_2\|^2} g_1^T \cdot g_2 \\ &= \|g_1\|^2 + \frac{(g_2^T \cdot g_1)^2}{\|g_2\|^2} - 2\frac{(g_2^T \cdot g_1)^2}{\|g_2\|^2} = \|g_1\|^2 - \frac{(g_2^T \cdot g_1)^2}{\|g_2\|^2} \\ &= \|g_1\|^2 - \frac{\|g_1\|^2 \|g_2\|^2 \cos^2\theta}{\|g_2\|^2} = \|g_1\|^2 - \|g_1\|^2 \cos^2\theta \\ &= \|g_1\|^2 \sin^2\theta \leq \|g_1\|^2 \end{aligned} \tag{6.11}$$

Property (3): $g_2^T \cdot g_1^O = 0$.

$$\begin{aligned} g_2^T \cdot g_1^O &= g_2^T \cdot (g_1 - \frac{g_2^T \cdot g_1}{\|g_2\|^2} g_2) = g_2^T \cdot g_1 - \frac{g_2^T \cdot g_1}{\|g_2\|^2} \|g_2\|^2 \\ &= g_2^T \cdot g_1 - g_2^T \cdot g_1 = 0 \end{aligned} \tag{6.12}$$

6.6.1 Gradient Combiner (GC) Convergence Proof

[131] discusses the requirements for a training algorithm to converge to its optimal answer. Here we will present a simplied version of Theorem 1 and Corollary 1 from [131].

Suppose that there are N training examples for a model with loss functions $L_1(w), \ldots, L_N(w)$ where w is the model parameter and w_0 is the initial model. Define $L(w) = \frac{1}{N}\sum_i L_i(w)$. Also assume that w^* is the optimal model where $L(w^*) \leq L(w)$ for all ws. A training algorithm is *pseudogradient* if:

- It is an iterative algorithm where $w_{i+1} = w_i - \alpha_i h_i$ where h_i is a random vector and α_i is a scalar.
- $\forall \epsilon \exists \delta : E(h_i)^T \cdot \nabla L(w) \geq \delta > 0$ where $L(w) \geq L(w^*) + \epsilon$ and w^* is the optimal model.
- $E(\|h_i\|^2) < C$ where C is a constant.
- $\forall i : \alpha_i \geq 0$, $\sum_i \alpha_i = \inf$, and $\sum_i \alpha_i^2 < \inf$.

The following Theorem is taken from [131].

Theorem 6.6.1. A pseudogradient training algorithm converges to the optimal model w^*.

As a reminder, $GC(g_1, g_2) = g_2 + g_1 - \frac{g_2^T \cdot g_1}{\|g_2\|^2} g_2$ and for k gradients, $GC(g_1, \ldots, g_k) = GC(g_k, \ldots, GC(g_3, GC(g_2, g_1)) \ldots)$.

Suppose $G(w_i) = \{\frac{\partial L_1}{\partial w}|_{w_i}, \dots, \frac{\partial L_N}{\partial w}|_{w_i}\}$ is a random variable distribution of the gradients at w_i.

Theorem 6.6.2. Suppose $h_i = GC(g_1, \dots, g_k)$ where $g_1, \dots, g_k$ are k independently chosen gradients from $G(w_i)$. h_i is pseudogradient.

Proof. To facilitate the proof of the pseudogradient properties of h_i, we rewrite GC formula as follows:

$$\begin{aligned} GC(g_1, g_2) &= g_2 + g_1 - \frac{g_2^T \cdot g_1}{\|g_2\|^2} g_2 = g_1 + g_2 - \frac{g_2 \cdot g_2^T}{\|g_2\|^2} g_1 \\ &= \Big(I - \frac{g_2 \cdot g_2^T}{\|g_2\|^2}\Big) g_1 + g_2 \end{aligned} \tag{6.13}$$

where $\frac{g_2 \cdot g_2^T}{\|g_2\|^2}$ is a rank-1 matrix.

First by induction, we prove that $E(GC(g_1, \dots, g_k))$ and $\nabla L(w_i)$ have a positive inner product.

Base of the induction: because g_1 and g_2 are independently chosen, $E(GC(g_1, g_2))$ can be calculated by:

$$\begin{aligned} E(GC(g_1, g_2)) &= E\Big(I - \frac{g_2 \cdot g_2^T}{\|g_2\|^2}\Big) E(g_1) + E(g_2) \\ &= \frac{1}{N} \sum_{a \in G(w_i)} \Big(I - \frac{a \cdot a^T}{\|a\|^2}\Big) \nabla L(w_i) + \nabla L(w_i) \end{aligned} \tag{6.14}$$

where the last equation comes from the fact that $E(g_j) = \nabla L(w_i)$ for any

$g_j \in G(w_i)$. Using the above formula, we have:

$$\begin{aligned}
&\nabla L(w_i)^T \cdot E(GC(g_1, g_2)) = \nabla L(w_i)^T \frac{1}{N} \sum_{a \in G(w_i)} \Big(I - \frac{a \cdot a^T}{\|a\|^2}\Big) \nabla L(w_i) \\
&+ \nabla L(w_i)^T \cdot \nabla L(w_i) = \frac{1}{N} \sum_{a \in G(w_i)} \Big(\|\nabla L(w_i)\|^2 \\
&- \nabla L(w_i)^T \frac{a \cdot a^T}{\|a\|^2} \nabla L(w_i) \Big) + \|\nabla L_{w_i}\|^2 \\
&= \frac{1}{N} \sum_{a \in G(w_i)} \Big(\|\nabla L(w_i)\|^2 - \|\nabla L(w_i)\|^2 \cos^2 \theta_a \Big) + \|\nabla L_{w_i}\|^2 \\
&= \frac{1}{N} \sum_{a \in G(w_i)} \Big(\|\nabla L(w_i)\|^2 \sin^2 \theta_a \Big) + \left\|\nabla L_(w_i)\right\|^2 \geq \left\|\nabla L_(w_i)\right\|^2
\end{aligned} \tag{6.15}$$

where θ_a is the angle between a and $\nabla L(w_i)$.

Induction step:

Now assume that $\nabla L(w_i)^T \cdot GC(g_1, \ldots, g_{l-1}) \geq \left\|\nabla L_(w_i)\right\|^2$. Also, assume that the random vector distribution that is generated by $GC(g_1, \ldots, g_{l-1})$ is X with a size of M. Therefore, for the induction step, we have:

$$\begin{aligned}
&\nabla L(w_i)^T \cdot E(GC(g_1, \ldots, g_{l-1}, g_l)) = \nabla L(w_i)^T \cdot \Bigg(E(g_l) \\
&+ E(GC(g_1, \ldots, g_{l-1})) - \frac{1}{M} \sum_{a \in X} \frac{a \cdot a^T}{\|a\|^2} E(g_1) \Bigg) = \|\nabla L(w_i)\|^2 \\
&+ \nabla L(w_i)^T \cdot E(GC(g_1, \ldots, g_{l-1})) - \frac{1}{M} \sum_{a \in X} \nabla L(w_i)^T \frac{a \cdot a^T}{\|a\|^2} \nabla L(w_i) \\
&= \nabla L(w_i)^T \cdot E(GC(g_1, \ldots, g_{l-1})) + \|\nabla L(w_i)\|^2 \\
&- \frac{1}{M} \sum_{a \in X} \|\nabla L(w_i)\|^2 \cos^2 \theta_a = \nabla L(w_i)^T \cdot E(GC(g_1, \ldots, g_{l-1})) \\
&+ \frac{1}{M} \sum_{a \in X} \|\nabla L(w_i)\|^2 \sin^2 \theta_a \geq \|\nabla L(w_i)\|^2
\end{aligned} \tag{6.16}$$

where θ_a is the angle between $\nabla L(w_i)$ and a and the last inequality is implied by the induction assumption. Therefore, the inner product of GC and $\nabla L(w_i)$ is positive in expectation as $\|\nabla L(w_i)\|^2$ is only zero at the optimal point.

Now we prove that norm of GC is bounded. We assume that norm of the gradients of $G(w_i)$ for all w_is are bounded.

$$\|GC(g_1, g_2)\|^2 = \left\|g_1^O + g_2\right\|^2 = \left\|g_1^O\right\|^2 + \|g_2\|^2 \leq \|g_1\|^2 + \|g_2\|^2 \tag{6.17}$$

where the last equality is implied by the Pythagorean theorem (property (3): g_1^O and g_2 are orthogonal) and the inequality is implied by $\left\|g_1^O\right\| \leq \|g_1\|$ (property (2)). This can be easily extended to GC for k gradients: $\|GC(g_1, \ldots, g_k)\|^2 \leq \sum_j \|g_j\|^2$.

Therefore, $E(\|GC(g_1, \ldots, g_k)\|^2) \leq \sum_j E(\|g_j\|^2) = kE(\|g_1\|)^2$. Therefore, norm of GC for k gradients is also bounded. Given that GC uses the same learning rate schedule as sequential SGD, the requirement for the learning rate of a pseudogradient training algorithm is already met. Thus, from Theorem 6.6.1, the gradients computed with GC moves the model to the optimal point. □

6.7 Experimental Evaluation

Section 6.7.1 outlines the evaluation methodology. We implement distributed Word2Vec and Vertex2Vec in our GraphAny2Vec framework, and we refer to these applications as GraphWord2Vec (GW2V) and GraphVertex2Vec

(GV2V) respectively. First, we compare these with the state-of-the-art third-party implementations (Section 6.7.2). We then analyze the impact of our Gradient Combiner (Section 6.7.3) and communication optimizations (Section 6.7.4). Our evaluation methodology is described in detail in Section 6.7.1.

6.7.1 Experimental Methodology

Hardware: All our experiments were conducted on the Stampede2 cluster at the Texas Advanced Computing Center using up to 32 Intel Xeon Platinum 8160 ("Skylake") nodes, each with 48 cores with clock rate 2.1Ghz, 192GB DDR4 RAM, and 32KB L1 data cache. Machines in the cluster are connected with a 100Gb/s Intel Omni-Path interconnect. Code is compiled with g++ 7.1 and MPI mvapich2/2.3.

Datasets: Table 6.1 lists the training datasets used for our evaluation: text datasets for Word2Vec and graph datasets for Vertex2Vec. These datasets have different vocabulary sizes (# unique words or vertices), total training corpus size (# occurrences of words or vertices), and sizes on disk. Prior Word2Vec and Vertex2Vec publications used the same datasets. The wiki (21GB) and Youtube (2.8GB) datasets are the largest text and graph datasets respectively. We used DeepWalk [128][4] for generating training corpus for Vertex2Vec by performing 10 random walks each of length 40 from all vertices of the graph. We limit our Word2Vec and Vertex2Vec evaluation to 32 and 16 hosts of Stampedes respectively because these datasets do not scale

[4] `https://github.com/phanein/deepwalk`

beyond that. We report the accuracy and the training (or execution) time for all frameworks on these datasets, excluding preprocessing time, as an average of three distinct runs.

Shared-memory third-party implementations: We evaluated the Skip-gram [113] (with negative sampling) training model for both Word2Vec and Vertex2Vec. We compared GraphWord2Vec (GW2V) with the state-of-the-art shared-memory Word2Vec implementations, the original C implementation (W2V) [113] as well as the more recent Gensim (GEN) [139] python implementation. We also compared our GraphVertex2Vec (GV2V) with the state-of-the-art shared-memory Vertex2Vec framework, DeepWalk [128] (both DeepWalk and Node2Vec [72] use Gensim's [139] Skip-gram model).

Distributed-memory third-party implementations: We compared GraphWord2Vec with the state-of-the-art distributed-memory Word2Vec from Microsoft's Distributed Machine Learning Toolkit (DMTK) [160], which is based on the parameter server model. The model is distributed among parameter server hosts. During execution, hosts acting as workers request the required model parameters from the servers and send model updates back to the servers. Each host in the cluster acts as both server and worker, and it is the only configuration possible. DMTK uses OpenMP for parallelization within a host (GraphWord2Vec uses Galois [118] for parallelization within a host). Both GraphWord2Vec and DMTK use MPI for communication between hosts. We modified DMTK to include a runtime option of configuring the number of synchronization rounds. DMTK uses averaging as the reduction

operation to combine the gradients. We refer to this as DMTK(AVG). We also implemented our Gradient Combiner in DMTK and we call this DMTK(GC). There are no prior distributed implementations of Vertex2Vec. Unless otherwise specified, GW2V and GV2V uses GC to combine gradients and use PullModel-Opt communication optimization.

Hyper-parameters: We used the hyper-parameters suggested by [113], unless otherwise specified: window size of 5, number of negative samples of 15, sentence length of 10K, threshold of 10^{-4} for Word2Vec and 0 for Vertex2Vec for down-sampling the frequent words, and vector dimensionality N of 200. All models were trained for 16 epochs. For distributed frameworks, GV2V, GW2V, and DMTK, we compared 2 gradient combining methods: Averaging (AVG) (default for distributed training of Any2Vec applications) and our novel Gradient Combiner (GC) method. Unless otherwise specified, GV2V, GW2V, and DMTK use the same number of synchronization rounds: 1 for 1 host, 3 for 2 hosts, 6 for 4 hosts, 12 for 8 hosts, 24 for 16 hosts, and 48 for 32 hosts. Note that the default for DMTK is 1 synchronization round for any number of hosts, but this yields very low accuracy, so we do not report these results.

Accuracy: In order to measure the accuracy of trained models of Word2Vec on different datasets, we used the analogical reasoning task outlined by original Word2Vec [113] paper. We evaluated the accuracy using scripts and question-words.txt provided by the Word2Vec code base[5]. Question-words.txt

[5] `https://github.com/tmikolov/word2vec`

Table 6.1: Datasets and their properties.

	Datasets	Vocabulary Words	Training Words	Size	Labels
Word2vec (Text)	**1-billion [38]**	399.0K	665.5M	3.7GB	N/A
	news [120]	479.3K	714.1M	3.9GB	N/A
	wiki [175]	2759.5K	3594.1M	21GB	N/A
Vertex2Vec (Graph)	**BlogCatalog [158]**	10.3K	4.1M	0.02GB	39
	Flickr [158]	80.5K	32.2M	0.18GB	195
	Youtube [158]	1138.5K	455.4M	2.8GB	47

Table 6.2: Word2Vec training time (hours) on a single host.

Dataset	W2V	GEM	DMTK(AVG)	GW2V(GC)
1-billion	4.24	4.39	4.21	3.98
news	4.45	4.66	4.28	4.51
wiki	20.49	OOM	25.43	22.34

consists of analogies such as "Athens" : "Greece" :: "Berlin" : ?, which are predicted by finding a vector x such that embedding vector(x) is closest to embedding vector("Athens") - vector("Greece") + vector("Berlin") according to the cosine distance. For this particular example the accepted value of x is "Germany". There are 14 categories of such questions, which are broadly divided into 2 main categories: (1) the syntactic analogies (such as "calm" : "calmly" :: "quick" : "quickly") and (2) the semantic analogies such as the country to capital city relationship. We report semantic, syntactic, and total accuracy averaged over all the 14 categories of questions. For Vertex2Vec , we measured *Micro-F1* and *Macro-F1* scores using scoring scripts provided by DeepWalk [128][6].

[6] `https://github.com/phanein/deepwalk/blob/master/example_graphs/scoring.py`

Table 6.3: Word2Vec accuracy (semantic, syntactic, and total) of different frameworks relative to W2V.

	Framework	**1-billion**	**news**	**wiki**
Semantic	W2V (1 Host)	75.86±0.07	70.79±0.54	79.10±0.31
	GEN (1 Host)	-0.22	-0.22	OOM
	DMTK (1 Host)	-13.79	-18.43	-7.46
	DMTK(AVG) (32 Hosts)	-57.36	-57.15	-34.39
	DMTK(GC) (32 Hosts)	-10.93	-17	-5.17
	GW2V (1 Host)	+0.07	-0.08	+0.26
	GW2V(AVG) (32 Hosts)	-7.00	-9.15	-4.03
	GW2V(GC) (32 Hosts)	+0.21	+0.07	-0.17
Syntactic	W2V (1 Host)	50.0±0.18	50.0±0.26	49.22±0.12
	GEN (1 Host)	-0.14	-0.12	OOM
	DMTK (1 Host)	-1.89	-0.67	-3.11
	DMTK(AVG) (32 Hosts)	-24.89	-25.11	-23.11
	DMTK(GC) (32 Hosts)	-3.56	-1.78	-1.44
	GW2V (1 Host)	-0.37	0.0	-0.12
	GW2V(AVG) (32 Hosts)	-4.89	-4.11	-7.55
	GW2V(GC) (32 Hostss)	+0.10	+0.11	+0.18
Total	W2V (1 Host)	72.36±0.21	69.21±0.42	74.10±0.42
	GEN (1 Host)	+0.0	-0.14	OOM
	DMTK (1 Host)	-11.65	-15.42	-3.03
	DMTK(AVG) (32 Hosts)	-51.29	-51.71	-32.03
	DMTK(GC) (32 Hosts)	-9.86	-14.78	-5.24
	GW2V (1 Host)	-0.14	-0.28	+0.1
	GW2V(AVG) (32 Hosts)	-6.79	-9.28	-5.17
	GW2V(GC) (32 Hostss)	+0.14	+0.29	-0.17

We implement distributed Word2Vec and Vertex2Vec in our GraphAny2Vec framework, and we refer to these applications as GraphWord2Vec(GW2V) and GrapVertex2Vec(GV2V) respectively. First, we compare these with the state-of-the-art third-party implementations (Section 6.7.2). We then analyze the impact of our Gradient Combiner (Section 6.7.3) and communication optimizations (Section 6.7.4). Our evaluation methodology is described in detail in Section 6.7.1.

6.7.2 Comparing With The State-of-The-Art

Word2Vec: We compare GW2V with distributed-memory implementation DMTK [160] and shared-memory implementations, W2V [113] and GEN [139]. Table 6.2 compares their training time on a single host. Figure 6.4 shows the speedup of both GW2V and DMTK on 32 hosts over W2V on 1 host. Note that averaging (AVG) and our Gradient Combiner (GC) methods are used to combine gradients during inter-host synchronization, so they have no impact on a single host.

Performance: We observe that for all datasets on a single host, the training time of GW2V is similar to that of W2V, GEN, and DMTK. GW2V scales up to 32 hosts and speeds up the training time by $\sim 13\times$ on average over 1 host. In comparison with distributed DMTK on 32 hosts, which uses parameter servers for synchronization , GW2V is $\sim 2\times$ faster on average for all datasets. Figure 6.4 also shows that there is negligible performance between using AVG and using GC to combine gradients in both DMTK and GW2V.

Training wiki using GW2V takes only **1.9 hours**, which saves **18.6 hours** and **1.5 hours** compared to training using W2V and DMTK respectively.

To understand the performance differences between DMTK and GW2V better, Figure 6.5 shows the breakdown of their training time into 3 phases: inspection, computation, and (non-overlapped) communication. Firstly, GW2V's inspection phase as well as serialization and de-serialization during synchronization are parallel using D-Galois [51, 118] parallel constructs and concurrent data-structures such as bit-vectors, work-lists, etc., whereas these phases are sequential in DMTK as it uses non-concurrent data-structures such as set and vector provided by the C++ standard template library. Moreover, in GW2V, hosts can update their masters in-place. This is not possible in DMTK as workers on each host have to fetch model parameters from servers on the same host to update, incurring overhead for additional copies. Secondly, DMTK communicates much higher volume ($\sim 3.5\times$) than GraphWord2Vec. GW2V memoizes the node IDs exchanged during inspection phase and sends only the updated values during broadcast and reduction. In contrast, DMTK sends the node IDs along with the updated values to the parameter servers during both broadcast and reduction. In addition, GraphWord2Vec inspection precisely identifies both the positive and negative samples required for the current round. DMTK, on the other hand, only identifies precise positive samples, and builds a pool for negative samples. During computation, negative samples are randomly picked from this pool. The entire pool is communicated from and to the parameter servers, although some of them may not be updated, leading

Table 6.4: Vertex2Vec training time (sec) of DeepWalk on 1 host vs. GraphVertex2Vec (GV2V) on 16 hosts.

Dataset	**DeepWalk**	**GV2V**	**Speedup**
BlogCatalog	115.3	28.8	4.0x
Flickr	976.7	183.1	5.3x
Youtube	11589.2	2226.2	5.2x

Table 6.5: Vertex2Vec accuracy (Macro F1 and Micro F1) of GV2V on 16 hosts relative to DeepWalk on 1 host.

		% Labeled Nodes	30%	60%	90%
Micro-F1	BlogCatalog	Deepwalk	34.0	37.2	38.4
		GV2V	-0.1	+0.1	+0.7
	Flickr	Deepwalk	38.6	40.4	41.1
		GV2V	+0.1	-0.1	-0.2
Macro-F1	BlogCatalog	Deepwalk	34.1	37.2	38.4
		GV2V	-0.3	+0.1	+0.7
	Flickr	Deepwalk	26.5	28.7	29.5
		GV2V	+0.1	-0.1	+0.3

to redundant communication.

Accuracy: Table 6.3 compares the accuracies (semantic, syntactic, and total) for all frameworks on 1 and 32 hosts relative to the accuracies achieved by W2V. On a single host, GW2V is able to achieve accuracies (semantic, syntactic and total) comparable to W2V. DMTK on a single host is less accurate due to implementation differences in the Skip-gram model training; DMTK only updates learning rate between mini-batches, whereas others continuously degrade learning rate, and DMTK uses a different strategy to choose negative samples as described earlier. On 32 hosts, DMTK(AVG) has terrible accuracy and GW2V(AVG) has poor accuracy. GC significantly improves the accuracies over AVG for both DMTK and GW2V. DMTK(GC) improves semantic by

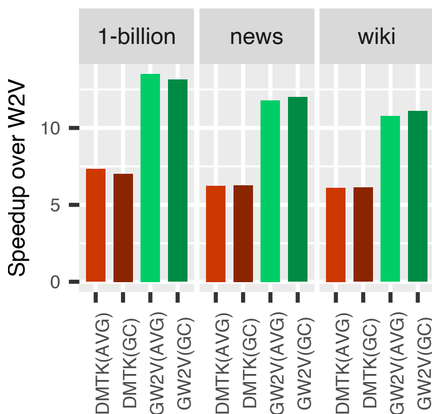


Figure 6.4: Speedup of DMTK and GW2V on 32 hosts over W2V on 1 host.

37.91%, syntactic by **22.09%**, and total by **34.79%** to match its own single host accuracy. GW2V(GC) improves all accuracies to match that of W2V.

Vertex2Vec: Table 6.4 compares the training time of DeepWalk [128] on a single host with our GV2V on 16 hosts. We observe that for all datasets, GraphVertex2Vec can train the model $\sim 4.8\times$ faster on average. Similar to GraphWord2Vec, this speedup does not come at the cost of the accuracy, as shown in Table 6.5, which shows the *Micro-F1* and *Macro-F1* score with 30%, 60%, and 90% labeled nodes.

Discussion: GraphAny2Vec significantly speeds up the training time for Word2Vec and Vertex2Vec applications by distributing the computation across the cluster without sacrificing the accuracy. Reduced training time also accelerates the process of improving the training algorithms as it allows application designers to make more end-to-end passes in a short duration of time.

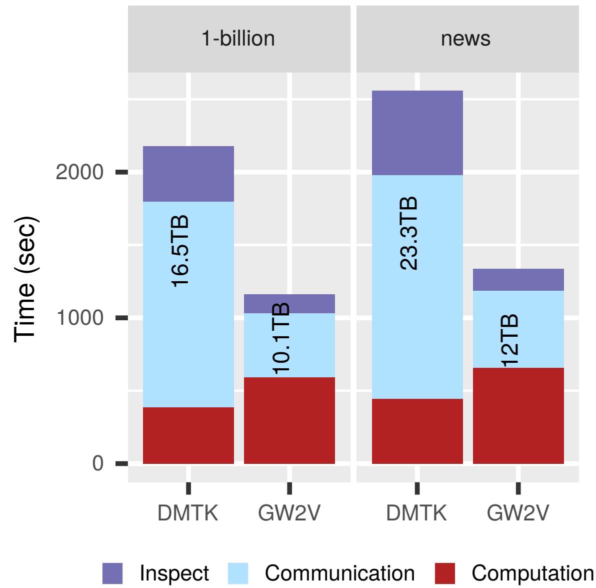


Figure 6.5: Breakdown of training time of DMTK(GC) and GW2V(GC) on 32 hosts.

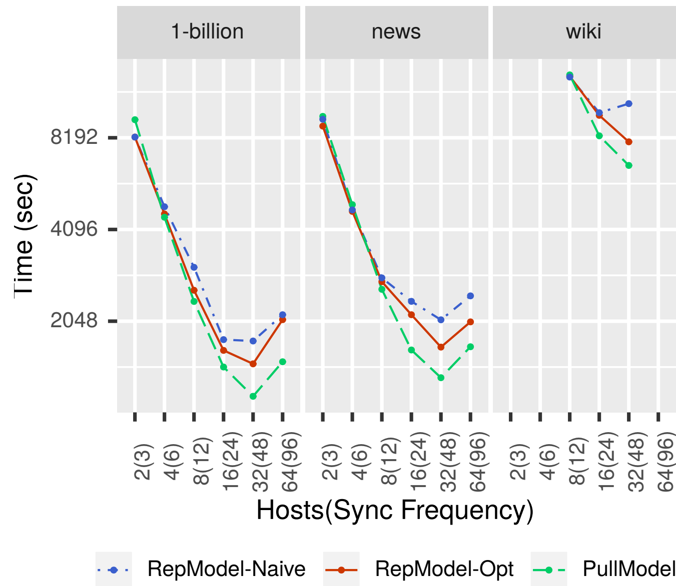


Figure 6.6: Strong scaling of GW2V (RMN: RepModel-Naive, RMO: RepModel-Opt, PMB: PullModel-Base, PMO: PullModel-Opt).

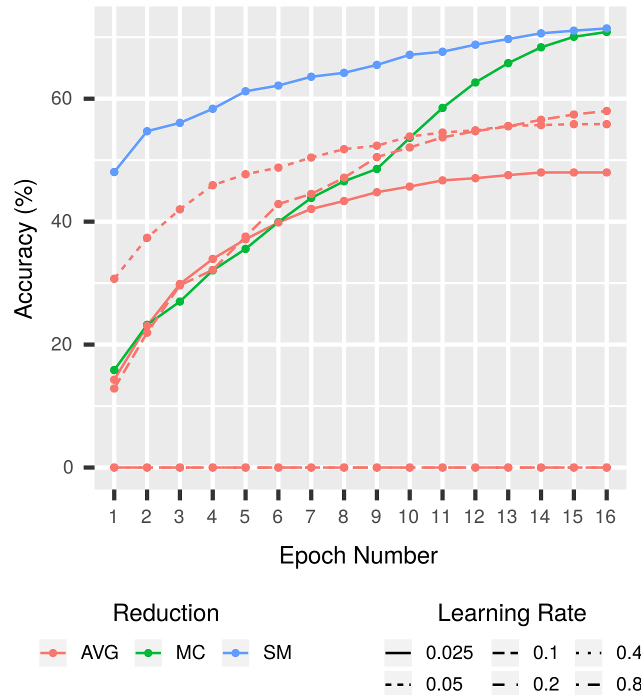


Figure 6.7: Accuracy of GW2V after each epoch for 1-billion dataset on 1 host (SM) and on 32 hosts using GC and AVG.

6.7.3 Impact of Gradient Combiner (GC)

If time were not an issue, all machine learning algorithms would run sequentially. A sequential SGD is simple to tune and converges fast. Unfortunately, it is slow. A point (x, y) on Figure 6.7 denotes the total accuracy (y) as a function of epoch (x). The blue line (SM) shows the accuracy of GW2V on a single shared-memory host. It clearly converges to a high accuracy quickly. In contrast, the red lines plot accuracy of distributed GW2V that uses averaging the gradients (AVG) with different learning rates on 32 hosts. The learning rate of 0.025 is the same as SM while the learning rate of 0.8 is 32 times larger. The former converges slowly while the latter does not

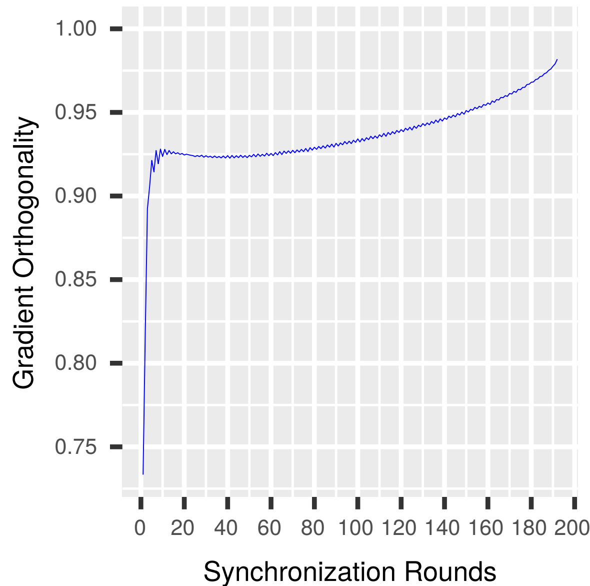


Figure 6.8: GW2V for 1-billion dataset on 32 hosts: % of gradients of a node that are orthogonal to each other in each epoch.

converge at all (accuracy is 0) because the learning rate is too large. Finally, the green line plots accuracy of distributed GW2V that uses GC and 0.025 as the learning rate on 32 hosts. GC has no problem meeting the accuracy of the sequential algorithm. In addition to providing the same accuracy as SM, it is 12× times faster on 32 hosts than SM. Not having to tune the learning rate and still getting accuracy at scale is a significant qualitative contribution of our work as tuning is a difficult task, in general.

In each round, we count the number of gradients that are being combined and the percentage of them that are orthogonal to each during combining. Figure 6.8 shows this percentage as function of the rounds. In later rounds, more and more gradients are orthogonal to each other. As explained in Section 6.5, GC is more effective when the gradients are orthogonal. This

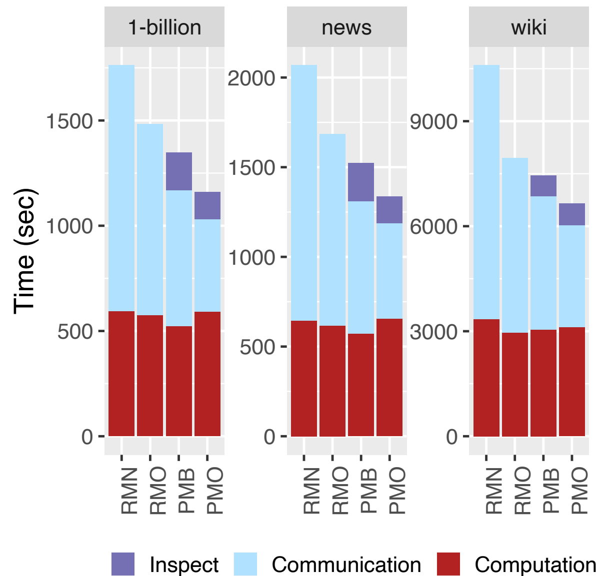


Figure 6.9: Breakdown of training time of GW2V with different communication schemes on 32 hosts.

is validated by the increase in accuracy for GC in later epochs in Figure 6.7.

Synchronization Rounds: Gradient Combiner (GC) improves the accuracies significantly but in order to get accuracies comparable to shared-memory implementations, the number of synchronization rounds in each epoch is an important knob to tune. We observe that accuracies improve as we increase the number of synchronization rounds within an epoch for both GC and AVG. Nonetheless, accuracies show more improvement for GC (for example, semantic: 3.07%, syntactic: 3.99% and total: 3.36% when synchronization frequency is increased from 12 to 48 on 32 hosts) as opposed to AVG, which shows very little change in accuracies with synchronization rounds. In general, we have observed that in order to maintain the desired accuracy, the synchronization frequency needs to be increased (roughly) linearly with the number of hosts.

We have followed this rule of thumb in all our experiments.

6.7.4 Impact of Communication Optimizations

Figure 6.6 shows the strong scaling of GraphWord2Vec with different communication optimizations (described in section 6.4.4): RepModel-Naive (RMN), RepModel-Opt (RMO), PullModel-Base (PMB), and PullModel-Opt (PMO). The 3 latter variants scale well up to 32 machines.

For 1-billion dataset, RepModel-Naive gives 4.7× speedup on 32 hosts over 2 hosts. RepModel-Opt, which uses D-Galois to only communicate the updated values for both reduction and broadcast, gives a speedup of 5.5× by reducing the communication volume. RepModel-Opt shows 16% improvement over RepModel-Naive on 32 hosts, showing that RepModel-Opt is able to exploit the sparsity in the communication. The benefits of RepModel-Opt over RepModel-Naive increases with the number of hosts for two main reasons: (a) synchronization frequency doubles with the number of hosts, thus communicating more data, and (b) as training data gets divided among hosts, sparsity in the updates increase.

PullModel-Base not only avoids replication of the model on all hosts (thus can fit bigger models), but helps reduce communication volume over RepModel-Opt by only broadcasting model parameters required by hosts for the next round. PullModel-Opt further reduces communication volume by taking into consideration the location of access as well: it only broadcasts embedding vector for sources and training vector for destinations. These benefits

come with an additional overhead of inspection phase before every synchronization round, but our evaluation shows that these overheads are offset by runtime improvements due to communication volume reduction. PullModel-Opt yields an average speedup of 8.1× on 32 hosts over 2 hosts and is $\sim 20.6\%$ on average faster than RepModel-Opt for all text datasets on 32 hosts.

Figure 6.9 shows the breakdown of the training time into inspection, computation, and communication time of the variants on 32 hosts. It is clear that all variants have similar computation time. RepModel-Opt communicates $\sim 2\times$ less communication volume on average as opposed to RepModel-Naive, thus improving the overall runtime. PullModel-Opt not only allows to train bigger models, it also further reduces communication volume by $\sim 11\%$ on average over RepModel-Opt by only broadcasting specific vectors to the proxies to be used in the next batch.

Summary: PullModel-Opt in GraphAny2Vec always performs better than the other variants by reducing the communication volume. These improvements are expected to grow as we scale to bigger datasets and number of hosts. Hence, PullModel-Opt not only allows one to train bigger models, but also gives the best performance.

6.8 Related Work

Many different types of models have been proposed in the past for estimating continuous representations of words, such as Latent Semantic Analysis (LSA) and Latent Dirichlet Allocation (LDA). However, distributed represen-

tations of words learned by neural networks are shown to perform significantly better than LSA [114, 193] and LDA is computationally very expensive on large data sets. Mikolov et al. [112] proposed two simpler model architectures for computing continuous vector representations of words from very large unstructured data sets, known as Continuous Bag-of-Words (CBOW) and Skip-gram (SG). These models removed the non-linear hidden layer and hence avoid dense matrix multiplications, which was responsible for most of the complexity in the previous models.

CBOW is similar to the feedforward Neural Net Language Models (NNLM) [17], where the non-linear hidden layer is removed and the projection layer is shared for all words. All words get projected into the same position and their vectors are averaged.

SG on the other hand unlike CBOW, instead of predicting the current word based on the context, tries to maximize classification of a word based on another word within a sentence. Later Mikolov et al. [113] further introduced several extensions, such as using hierarchical softmax instead of full softmax, negative sampling, subsampling of frequent words, etc., to SG model that improves both the quality of the vectors and the training speed.

Our work adapts the algorithm from this later work [113] for distribution. This work, together with many current implementations [139] are designed to run on a single machine but utilizing multi-threaded parallelism. Our work is motivated by the fact that these popularly used implementations take days or even weeks to train on large training corpus. Prior works on

distributing Word2Vec either use synchronous data-parallelism [85, 176, 177] or parameter-server style asynchronous data parallelism [160]. However, they perform communication after every mini-batch, which is prohibitively expensive in terms of network bandwidth. Our design was motivated by the need to use commodity machines and network available on public clouds. Our approach communicates infrequently and uses our novel Gradient Combiner to overcome the resulting staleness.

Ordentlich et al. [122] propose a different method designed for models that do not fit in the memory of a single machine. They partition the model vertically with each machine containing part of the embedding and training vector for each word. These partitions compute partial dot products locally but communicate to compute global dot products. For all the publicly available benchmarks we could find, the models fit in the memory in our machines. Nevertheless, our design allows for horizontal partitioning of large models if such a need arises in the future.

Chapter 7

Conclusion

Graph analytics systems are used in a wide variety of applications including health care, electronic circuit design, machine learning, and cybersecurity. Graph analytics systems must handle very large graphs such as the Facebook friends graph, which has more than a billion nodes and 200 billion edges. Single machines used for shared-memory graph analytics are limited in main memory and compute resources.

In order to break free from the main memory limitation, this dissertation looks at the new opportunities for graph analytics on massive datasets on a single machine brought by a new kind of byte-addressable memory technology with higher density and lower cost than DRAM such as intel Optane DC Persistent Memory. This enables the design of affordable systems that support up to 6TB of randomly accessible memory. In Chapter 2, we present key runtime and algorithmic principles to consider when performing graph analytics on massive datasets on Optane DC Persistent Memory as well as highlight ideas that apply to graph analytics on all large-memory platforms.

In order to provide more compute resource, distributed-memory clusters with sufficient memory and computation power are required for process-

ing of these graphs. In distributed graph analytics, the graph is partitioned among the machines in a cluster, and communication between partitions is implemented using a substrate like MPI. However, programming distributed-memory systems are not easy and the recent trend towards the processor heterogeneity has added to this complexity. To simplify the programming of graph applications on such platforms, this dissertation (Chapter 3) first presents a compiler called Abelian that translates shared-memory descriptions of graph algorithms written in the Galois programming model into efficient code for distributed-memory platforms with heterogeneous processors.

An important runtime parameter to the compiler-generated distributed code is the partitioning policy. Chapter 4 presents an experimental study of partitioning strategies for distributed work-efficient graph analytics applications on different CPU architecture clusters at large scale (up to 256 machines). Based on the study we present a simple rule of thumb to select among myriad policies.

Another challenge of distributed graph analytics that we address in this dissertation is to deal with machine fail-stop failures, which is an important concern especially for long-running graph analytics applications on large clusters. This dissertation presents a novel communication and synchronization substrate called Phoenix that leverages the algorithmic properties of graph analytics applications to recover from faults with zero overheads during fault-free execution as described in Chapter 5 and show that Phoenix is 24x faster than previous state-of-the-art systems.

Finally, in Chapter 6, we show that our distributed graph analytics infrastructure can be used for a new domain of applications, in particular, embedding algorithms such as Word2Vec. Word2Vec trains the vector representations of words (also known as word embeddings) on large text corpus and resulting vector embeddings have been shown to capture semantic and syntactic relationships among words. Other examples include Node2Vec, Code2Vec, Sequence2Vec, etc (collectively known as Any2Vec) with a wide variety of uses. We formulate the training of such applications as a graph problem and present GraphAny2Vec, a distributed Any2Vec training framework that leverages the state-of-the-art distributed heterogeneous graph analytics infrastructure developed in this dissertation to scale Any2Vec training to large distributed clusters. GraphAny2Vec also demonstrates a novel way of combining model gradients during training, which allows it to scale without losing accuracy.

Bibliography

[1] http://www.graph500.org.

[2] Lustre file system, 2018.

[3] Pittsburgh Supercomputing Center (PSC), 2018.

[4] Texas Advanced Computing Center (TACC), The University of Texas at Austin, 2018.

[5] Persistent Memory Programming, pmem.io, 2019.

[6] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. Streaming Graph Partitioning: An Experimental Study. *PVLDB*, 11(11), 2018.

[7] Amine Abou-Rjeili and George Karypis. Multilevel Algorithms for Partitioning Power-law Graphs. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 124–124, Washington, DC, USA, 2006. IEEE Computer Society.

[8] Alekh Agarwal, Oliveier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *Journal of Machine Learning Research*, 15:1111–1133, 2014.

[9] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS '04, pages 277–286, New York, NY, USA, 2004. ACM.

[10] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. Squeezing out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 125–137, Santa Clara, CA, July 2017. USENIX Association.

[11] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019.

[12] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind logging. *Proc. VLDB Endow.*, 10(4):337–348, November 2016.

[13] Joshua Auerbach, David F. Bacon, Ioana Burcea, Perry Cheng, Stephen J. Fink, Rodric Rabbah, and Sunil Shukla. A compiler and runtime for heterogeneous computing. DAC, 2012.

[14] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[15] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[16] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 235–248, New York, NY, USA, 2017. ACM.

[17] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, March 2003.

[18] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[19] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[20] Paolo Boldi and Sebastiano Vigna. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[21] Paolo Boldi and Sebastiano Vigna. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[22] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[23] E. G. Boman, K. D. Devine, and S. Rajamanickam. Scalable matrix computations on large scale-free graphs using 2D graph partitioning. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, Nov 2013.

[24] George Bosilca, Remi Delmas, Jack Dongarra, and Julien Langou. Algorithm based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410 – 416, 2009.

[25] Leon Bottou. *Stochastic Gradient Descent Tricks*, volume 7700 of *Lecture Notes in Computer Science (LNCS)*, pages 430–445. Springer, neural networks, tricks of the trade, reloaded edition, January 2012.

[26] Léon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20 (NIPS 2007)*, pages 161–168. NIPS Foundation (http://books.nips.cc), 2008.

[27] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization Methods for Large-Scale Machine Learning. *arXiv e-prints*, Jun 2016.

[28] Florian Bourse, Marc Lelarge, and Milan Vojnovic. Balanced Graph Edge Partition. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 1456–1465, New York, NY, USA, 2014. ACM.

[29] Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Recent advances in checkpoint/recovery systems. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 282–282, Washington, DC, USA, 2006. IEEE Computer Society.

[30] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of mpi programs. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 84–94, New York, NY, USA, 2003. ACM.

[31] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill.

C3: A system for automating application-level checkpointing of mpi programs. pages 357–373, 05 2004.

[32] Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. CGO, 2016.

[33] Aydin Buluc and John R Gilbert. The Combinatorial BLAS: Design, Implementation, and Applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, November 2011.

[34] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in apache flink®: Consistent stateful distributed stream processing. *PVLDB*, 10(12):1718–1729, 2017.

[35] Ümit V. Çatalyürek, Cevdet Aykanat, and Bora Uçar. On Two Dimensional Sparse Matrix Partitioning: Models, Methods, and a Recipe. *SIAM J. Sci. Comput.*, 32(2):656–683, February 2010.

[36] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. *R-MAT: A Recursive Model for Graph Mining*, pages 442–446.

[37] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[38] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, and Phillipp Koehn. One billion word benchmark for measuring progress in statistical language modeling. *CoRR*, abs/1312.3005, 2013.

[39] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 1:1–1:15, New York, NY, USA, 2015. ACM.

[40] Youmin Chen, Jiwu Shu, Jiaxin Ou, and Youyou Lu. Hinfs: A persistent memory file system with both buffering and direct-access. *ACM Trans. Storage*, 14(1):4:1–4:30, April 2018.

[41] Zizhong Chen. Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 167–176, New York, NY, USA, 2013. ACM.

[42] Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Fault tolerant high performance computing by a coding approach. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 213–223, New York, NY, USA, 2005. ACM.

[43] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.

[44] Ge-Ming Chiu and Cheng-Ru Young. Efficient rollback-recovery technique in distributed computing systems. *IEEE Trans. Parallel Distrib. Syst.*, 7(6):565–577, June 1996.

[45] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[46] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[47] Intel® Corporation. Intel® vtune™ platform profiler analysis, 2019.

[48] Intel® Corporation. Intel® vtune™ amplifier, 2019.

[49] Intel® Corporation. Ipmctl Intel® Optane™ DC Persistent Memory, 2019.

[50] Hoang-Vu Dang, Roshan Dathathri, Gurbinder Gill, Alex Brooks, Nikoli Dryden, Andrew Lenharth, Loc Hoang, Keshav Pingali, and Marc Snir. A lightweight communication runtime for distributed graph analytics. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

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

[52] Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. Phoenix: A substrate for resilient distributed graph analytics. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 615–630, New York, NY, USA, 2019. ACM.

[53] Teresa Davies, Christer Karlsson, Hui Liu, Chong Ding, and Zizhong Chen. High performance linpack benchmark: A fault tolerant implementation without checkpointing. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 162–171, New York, NY, USA, 2011. ACM.

[54] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[55] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04*, 2004.

[56] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 393–404, New York, NY, USA, 2018. ACM.

[57] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, November 1974.

[58] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[59] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002.

[60] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. Comput.*, 41(5):526–531, May 1992.

[61] Yun Gao, Wei Zhou, Jizhong Han, Dan Meng, Zhang Zhang, and Zhiyong Xu. An Evaluation and Analysis of Graph Processing Frameworks on Five Key Issues. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 11:1–11:8, New York, NY, USA, 2015. ACM.

[62] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. PACT, 2012.

[63] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Andrew Lenharth, and Keshav Pingali. Abelian: A compiler for graph analytics on distributed, heterogeneous platforms. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018: Parallel Processing*, pages 249–264, Cham, 2018. Springer International Publishing.

[64] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single machine graph analytics on massive datasets using intel optane dc persistent memory. *Proc. VLDB Endow.*, 13(8), 2020.

[65] Gurbinder Gill, Roshan Dathathri, Loc Hoang, and Keshav Pingali. A Study of Partitioning Policies for Graph Analytics on Large-scale Distributed Platforms. volume 12 of *PVLDB*, 2018.

[66] Gurbinder Gill, Roshan Dathathri, Loc Hoang, and Keshav Pingali. A study of partitioning policies for graph analytics on large-scale distributed platforms. *Proc. VLDB Endow.*, 12(4):321–334, December 2018.

[67] Gurbinder Gill, Roshan Dathathri, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Distributed training of embeddings using graph analytics, 2019.

[68] Apache Giraph. http://giraph.apache.org/, 2013.

[69] L. A. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka. Distributed diskless checkpoint for large scale systems. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 63–72, May 2010.

[70] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[71] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[72] Aditya Grover and Jure Leskovec. Node2vec: Scalable feature learning for networks. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 855–864, New York, NY, USA, 2016. ACM.

[73] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[74] Ruining He and Julian McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. WWW, 2016.

[75] Loc Hoang, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. CuSP: A Customizable Streaming Edge Partitioner for Distributed Graph Analytics. In *Proceedings of the 33rd IEEE International Parallel and Distributed Processing Symposium*, IPDPS 2019, 2019.

[76] Loc Hoang, Matteo Pontecorvi, Roshan Dathathri, Gurbinder Gill, Bozhi You, Keshav Pingali, and Vijaya Ramachandran. A round-efficient distributed betweenness centrality algorithm. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'19)*, PPoPP, 2019.

[77] Mark Frederick Hoemmen and Michael Allen Heroux. Fault-tolerant iterative methods. Technical report, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2011.

[78] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. ASPLOS, 2012.

[79] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. PGX.D: A Fast Distributed Graph Processing Engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 58:1–58:12, New York, NY, USA, 2015. ACM.

[80] Imranul Hoque and Indranil Gupta. LFGraph: Simple and Fast Distributed Graph Analytics. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, TRIOS '13, pages 9:1–9:17, New York, NY, USA, 2013. ACM.

[81] Kuang-Hua Huang and Abraham. Algorithm based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6):518–528, June 1984.

[82] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[83] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R.

Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[84] Nilesh Jain, Guangdeng Liao, and Theodore L. Willke. GraphBuilder: Scalable Graph ETL Framework. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 4:1–4:6, New York, NY, USA, 2013. ACM.

[85] S. Ji, N. Satish, S. Li, and P. K. Dubey. Parallelizing word2vec in shared and distributed memory. *IEEE Transactions on Parallel and Distributed Systems*, 30(9):2090–2100, Sep. 2019.

[86] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. A distributed multi-gpu system for fast graph processing. *Proc. VLDB Endow.*, November 2017.

[87] Sang Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. Bigsparse: High-performance external graph analytics. *CoRR*, 2017.

[88] George Karypis and Vipin Kumar. Parallel Multilevel K-way Partitioning Scheme for Irregular Graphs. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society.

[89] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*,

20(1):359–392, December 1998.

[90] George Karypis and Vipin Kumar. Multilevel K-way Hypergraph Partitioning. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, pages 343–348, New York, NY, USA, 1999. ACM.

[91] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 169–182, New York, NY, USA, 2013. ACM.

[92] Mijung Kim and K. Selçuk Candan. SBV-Cut: Vertex-cut Based Graph Partitioning Using Structural Balance Vertices. *Data Knowl. Eng.*, 72:285–303, February 2012.

[93] Dhananjay Kimothi, Akshay Soni, Pravesh Biyani, and James M. Hogan. Distributed representations for biological sequence analysis. *CoRR*, abs/1608.05949, 2016.

[94] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 399–411, New York, NY, USA, 2016. ACM.

[95] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, Jan 1987.

[96] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[97] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[98] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[99] Monica S. Lam, Stephen Guo, and Jiwon Seo. SociaLite: Datalog Extensions for Efficient Social Network Analysis. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 278–289, Washington, DC, USA, 2013. IEEE Computer Society.

[100] Michael LeBeane, Shuang Song, Reena Panda, Jee Ho Ryoo, and Lizy K. John. Data Partitioning Strategies for Graph Workloads on Heterogeneous Clusters. In *Proceedings of the International Conference for*

High Performance Computing, Networking, Storage and Analysis, SC '15, pages 56:1–56:12, New York, NY, USA, 2015. ACM.

[101] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Parallel Graph Analytics. *Commun. ACM*, 59(5):78–87, April 2016.

[102] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker Graphs: An Approach to Modeling Networks. *J. Mach. Learn. Res.*, 11:985–1042, March 2010.

[103] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 5(8):716–727, 2012.

[104] David E. Lowell, Subhachandra Chandra, and Peter M. Chen. Exploring failure transparency and the limits of generic recovery. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, Berkeley, CA, USA, 2000. USENIX Association.

[105] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 527–543, New York, NY, USA, 2017. ACM.

[106] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. ACM SIGMOD Intl Conf. on Management of Data*, SIGMOD '10, pages 135–146, 2010.

[107] Jasmina Malicevic, Subramanya Dulloor, Narayanan Sundaram, Nadathur Satish, Jeff Jackson, and Willy Zwaenepoel. Exploiting NVM in Large-scale Graph Analytics. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15, pages 2:1–2:9, New York, NY, USA, 2015. ACM.

[108] D. Manivannan and M. Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):703–713, July 1999.

[109] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 499–512, New York, NY, USA, 2017. ACM.

[110] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. Web Data Commons - Hyperlink Graphs, 2012.

[111] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. Graph Structure in the Web — Revisited: A Trick of the Heavy Tail. In

Proceedings of the 23rd International Conference on World Wide Web, WWW '14 Companion, pages 427–432, New York, NY, USA, 2014. ACM.

[112] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space.

[113] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 3111–3119, USA, 2013. Curran Associates Inc.

[114] Tomas Mikolov, Scott Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT-2013)*. Association for Computational Linguistics, May 2013.

[115] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov 2010.

[116] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Data-driven versus Topology-driven Irregular Computations on GPUs. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '13, London, UK, 2013. Springer-Verlag.

[117] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant Software Distributed Shared Memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 291–305, Berkeley, CA, USA, 2015. USENIX Association.

[118] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[119] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP, pages 456–471, 2013.

[120] Ninth Workshop on Statistical Machine Translation.

[121] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[122] Erik Ordentlich, Lee Yang, Andy Feng, Peter Cnudde, Mihajlo Grbovic, Nemanja Djuric, Vladan Radosavljevic, and Gavin Owens. Network-efficient distributed word2vec training system for large vocabularies. In *Proceedings of the 25th ACM International Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*, pages 1139–1148, 2016.

[123] Carlos Pachajoa and Wilfried N. Gansterer. On the resilience of conjugate gradient and multigrid methods to node failures. In *Euro-Par 2017: Parallel Processing Workshops*, pages 569–580, Cham, 2018. Springer International Publishing.

[124] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 1–19, New York, NY, USA, 2016. ACM.

[125] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D. Owens. Multi-gpu graph analytics. IPDPS, May 2017.

[126] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. System evaluation of the intel optane byte-addressable nvm. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, pages 304–315, New York, NY, USA, 2019. ACM.

[127] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 701–710, New York, NY, USA, 2014. ACM.

[128] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 701–710, New York, NY, USA, 2014. ACM.

[129] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. HDRF: Stream-Based Partitioning for Power-Law Graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, pages 243–252, New York, NY, USA, 2015. ACM.

[130] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The TAO of parallelism in algorithms. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, PLDI '11, pages 12–25, 2011.

[131] Boris Polyak and Y.Z. Tsypkin. Pseudogradient adaptation and training algorithms. *Automation and Remote Control*, 34:377–397, 01 1973.

[132] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 293–306, Berkeley, CA, USA, 2010. USENIX Association.

[133] D. Presser, L. C. Lung, and M. Correia. Greft: Arbitrary fault-tolerant distributed graph processing. In *2015 IEEE International Congress on Big Data*, pages 452–459, June 2015.

[134] The Lemur Project. The ClueWeb12 Dataset, 2013.

[135] Dimitris Prountzos, Roman Manevich, and Keshav Pingali. Synthesizing parallel graph programs via automated planning. In *Programming Language Design and Implementation*, PLDI'15, 2015.

[136] Mayank Pundir, Luke M. Leslie, Indranil Gupta, and Roy H. Campbell. Zorro: Zero-cost reactive failure recovery in distributed graph processing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 195–208, New York, NY, USA, 2015. ACM.

[137] Feng Qin, Joseph Tucek, Yuanyuan Zhou, and Jagadeesan Sundaresan. Rx: Treating Bugs As Allergies - a Safe Method to Survive Software Failures. *ACM Trans. Comput. Syst.*, 25(3), August 2007.

[138] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q.

Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701. Curran Associates, Inc., 2011.

[139] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. http://is.muni.cz/publication/884893/en.

[140] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: A compiler and runtime for heterogeneous systems. SOSP, 2013.

[141] Amitabha Roy, 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*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[142] Semih Salihoglu and Jennifer Widom. GPS: A Graph Processing System. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 22:1–22:12, New York, NY, USA, 2013. ACM.

[143] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Vishal Sahay, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The lam/mpi checkpoint/restart framework: System-initiated checkpointing. *The International Journal of High Performance Computing Applications*, 19(4):479–493, 2005.

[144] Piyush Sao, Oded Green, Chirag Jain, and Richard Vuduc. A self-correcting connected components algorithm. In *Proceedings of the ACM Workshop on Fault-Tolerance for HPC at Extreme Scale*, FTXS '16, pages 9–16, New York, NY, USA, 2016. ACM.

[145] Piyush Sao and Richard Vuduc. Self-stabilizing iterative solvers. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '13, pages 4:1–4:8, New York, NY, USA, 2013. ACM.

[146] Sebastian Schelter, Stephan Ewen, Kostas Tzoumas, and Volker Markl. "all roads lead to rome": Optimistic recovery for distributed iterative data processing. In *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management*, CIKM '13, pages 1919–1928, New York, NY, USA, 2013. ACM.

[147] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, Oct 2010.

[148] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM.

[149] Yanyan Shen, Gang Chen, H. V. Jagadish, Wei Lu, Beng Chin Ooi, and Bogdan Marius Tudor. Fast failure recovery in distributed graph

processing systems. *Proc. VLDB Endow.*, 8(4):437–448, December 2014.

[150] Zhan Shi, Junhao Li, Pengfei Guo, Shuangshuang Li, Dan Feng, and Yi Su. Partitioning Dynamic Graph Asynchronously with Distributed FENNEL. *Future Gener. Comput. Syst.*, 71(C):32–42, June 2017.

[151] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, 2013.

[152] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri. Partitioning Trillion-Edge Graphs in Minutes. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 646–655, May 2017.

[153] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 297–310, New York, NY, USA, 2015. ACM.

[154] Isabelle Stanton and Gabriel Kliot. Streaming Graph Partitioning for Large Distributed Graphs. In *Proceedings of the 18th ACM SIGKDD In-*

ternational Conference on Knowledge Discovery and Data Mining, KDD '12, pages 1222–1230, New York, NY, USA, 2012. ACM.

[155] Dan Stanzione, Bill Barth, Niall Gaffney, Kelly Gaither, Chris Hempel, Tommy Minyard, S. Mehringer, Eric Wernert, H. Tufo, D. Panda, and P. Teller. Stampede 2: The Evolution of an XSEDE Supercomputer. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, PEARC17, pages 15:1–15:8, New York, NY, USA, 2017. ACM.

[156] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. Shortcutting label propagation for distributed connected components. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, WSDM '18, pages 540–546, New York, NY, USA, 2018. ACM.

[157] Rob Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, August 1985.

[158] Lei Tang and Huan Liu. Relational learning via latent social dimensions. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 817–826, New York, NY, USA, 2009. ACM.

[159] Xiongchao Tang, Jidong Zhai, Bowen Yu, Wenguang Chen, and Weimin Zheng. Self-checkpoint: An in-memory checkpoint method using less

space and its practice on fault-tolerant hpl. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 401–413, New York, NY, USA, 2017. ACM.

[160] Distributed Machine Learning Toolkit. `http://www.dmtk.io/`.

[161] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D Peterson, et al. XSEDE: accelerating scientific discovery. *Computing in Science & Engineering*, 16(5):62–74, 2014.

[162] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, Ralph Roskies, J. Ray Scott, and Nancy Wilkins-Diehr. Xsede: Accelerating scientific discovery. *Computing in Science and Engineering*, 16(5):62–74, Sept-Oct 2014.

[163] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14, pages 333–342, New York, NY, USA, 2014. ACM.

[164] Nitesh Upadhyay, Parita Patel, Unnikrishnan Cheramangalath, and Y. N. Srikant. Large scale graph processing in a distributed environment. In *Euro-Par 2017 Parallel Processing Workshops*, 2018.

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

[166] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1541–1555, New York, NY, USA, 2018. ACM.

[167] Shiv Verma, Luke M. Leslie, Yosub Shin, and Indranil Gupta. An experimental comparison of partitioning strategies in distributed graph processing. *Proc. VLDB Endow.*, 10(5):493–504, January 2017.

[168] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[169] Keval Vora. LUMOS: Dependency-Driven Disk-based Graph Processing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 429–442, Renton, WA, July 2019. USENIX Association.

[170] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms Using a Relaxed Consistency Based DSM. In *Proceedings of the 2014 ACM International*

Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '14, pages 861–878, New York, NY, USA, 2014. ACM.

[171] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. Coral: Confined recovery in distributed asynchronous graph processing. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 223–236, New York, NY, USA, 2017. ACM.

[172] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan. Replication-based fault-tolerance for large-scale graph processing. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 562–573, June 2014.

[173] R. Wang, E. Yao, M. Chen, G. Tan, P. Balaji, and D. Buntinas. Building algorithmically nonstop fault tolerant mpi programs. In *2011 18th International Conference on High Performance Computing*, pages 1–9, Dec 2011.

[174] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. An early evaluation of intel's optane dc persistent memory module and its impact on high-performance scientific applications. In *Proceedings of the International Conference for High Per-*

formance Computing, Networking, Storage and Analysis, SC '19, pages 76:1–76:19, New York, NY, USA, 2019. ACM.

[175] Wikimedia. `https://dumps.wikimedia.org/enwiki/latest/`.

[176] word2vec implementation in Deeplearning4j. `https://deeplearning4j.org/docs/latest/deeplearning4j-nlp-word2vec`.

[177] word2vec implementation in Spark. `https://spark.apache.org/docs/latest/mllib-feature-extraction.html`.

[178] Kai Wu, Frank Ober, Shari Hamlin, and Dong Li. Early evaluation of intel optane non-volatile memory with HPC I/O workloads. *CoRR*, abs/1708.02199, 2017.

[179] Panruo Wu and Zizhong Chen. Ft-scalapack: Correcting soft errors on-line for scalapack cholesky, qr, and lu factorization routines. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 49–60, New York, NY, USA, 2014. ACM.

[180] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. Tux: Distributed Graph Computation for Machine Learning. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 669–682, Boston, MA, 2017. USENIX Association.

[181] Reynold S. Xin, 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*, GRADES '13, 2013.

[182] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, 2016. USENIX Association.

[183] Fan Yang, Ming Wu, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. GraM: Scaling Graph Computation to the Trillions. In *SoCC*. ACM – Association for Computing Machinery, August 2015.

[184] E. Yao, R. Wang, M. Chen, G. Tan, and N. Sun. A case study of designing efficient algorithm-based fault tolerant application for exascale parallelism. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 438–448, May 2012.

[185] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, September 1974.

[186] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for

in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[187] Mikhail Zarubin, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. Efficient compute node-local replication mechanisms for nvram-centric data structures. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, DAMON '18, pages 7:1–7:9, New York, NY, USA, 2018. ACM.

[188] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 183–193, New York, NY, USA, 2015. ACM.

[189] Yu Zhang, Yanbing Liu, Jing Yu, Ping Liu, and Li Guo. VSEP: A Distributed Algorithm for Graph Edge Partitioning. In *Proceedings of the ICA3PP International Workshops and Symposiums on Algorithms and Architectures for Parallel Processing - Volume 9532*, pages 71–84, Berlin, Heidelberg, 2015. Springer-Verlag.

[190] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proc. ACM Program. Lang.*, 2(OOPSLA):121:1–121:30, October 2018.

[191] G. Zheng, Xiang Ni, and L. V. Kalé. A scalable double in-memory checkpoint and restart scheme towards exascale. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*, pages 1–6, June 2012.

[192] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhiming Ma, and Tie-Yan Liu. Asynchronous stochastic gradient descent with delay compensation for distributed deep learning. *CoRR*, abs/1609.08326, 2016.

[193] Alisa Zhila, Scott Wen-tau Yih, Geoffrey Zweig, Chris Meek, and Tomas Mikolov. Combining heterogeneous models for measuring relational similarity. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT-2013)*. Association for Computational Linguistics, May 2013.

[194] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on gpus. *IEEE TPDS*, 2014.

[195] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-centric Distributed Graph Processing System. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 301–316, Berkeley, CA, USA, 2016. USENIX Association.

[196] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 375–386, Berkeley, CA, USA, 2015. USENIX Association.