T2R2 東京工業大学リサーチリポジトリ
Tokyo Tech Research Repository

論文 / 著書情報
Article / Book Information

題目(和文)	局所性を考慮した大規模グラフ解析向けグラフデータストア
Title(English)	Locality-aware Graph Data Store for Large-scale Graph Analytics
著者(和文)	岩渕圭太
Author(English)	Keita Iwabuchi
出典(和文)	学位:博士(理学), 学位授与機関:東京工業大学, 報告番号:甲第10435号, 授与年月日:2017年3月26日, 学位の種別:課程博士, 審査員:松岡 聡,南出 靖彦,渡辺 治,遠藤 敏夫,脇田 建
Citation(English)	Degree:Doctor (Science), Conferring organization: Tokyo Institute of Technology, Report number:甲第10435号, Conferred date:2017/3/26, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Powered by T2R2 (Tokyo Institute Research Repository)

TOKYO INSTITUTE OF TECHNOLOGY

Locality-aware Graph Data Store for Large-scale Graph Analytics

by

Keita Iwabuchi

A thesis submitted in partial fulfillment for the

degree of Doctor of Science

in

Mathematical and Computing Sciences

in the

Graduate School of Information Science and Engineering

Committee Chair in charge:

Professor Satoshi Matsuoka

February, 2017

©Copyright 2017 Keita Iwabuchi
All Rights Reserved

TOKYO INSTITUTE OF TECHNOLOGY

Abstract

Graduate School of Information Science and Engineering

Doctor of Science
in
Mathematical and Computing Sciences

Locality-aware Graph Data Store for Large-scale Graph Analytics

by Keita Iwabuchi

Big data processing brings us many challenges attributed to not only its volume but also to the emergence of a new paradigm; that is, analyzing the data to discover knowledge, to understand behaviors, and to mine for patterns accompanied with complex memory access patterns on large volume of data. At the same time, demands for large-scale graph analytics has risen as an important kernel for high-performance computing (HPC) applications in various domains, such as WWW and social network analysis, network security, artificial intelligence and genomic analysis. Meanwhile, the interest in non-volatile random-access memory (NVRAM) such as NAND flash, phase change memory (PCM) and resistive RAM (ReRAM) has risen due to the cost and high power consumption of DRAM. However, large-scale graph analytics often presents challenging data-intensive workloads because of unstructured and random memory access patterns. Therefore, designing locality-aware data stores for graph analytics is an extremely important key factor to enable high-performance graph analytics.

To address the issues above, we explore techniques and designs of large-scale graph data stores from the perspective of static and dynamic graph analytics.

First, we explore important techniques for out-of-core static graph stores by developing a high performance out-of-core breadth-first search (BFS) implementation. Specifically, we propose NETALX, an extremely high performance BFS implementation using NVRAM for Hybrid BFS algorithm which devises the arrangement of graph data on DRAM and NVRAM to improve data locality (reduce

the number of accesses to NVRAM) and sequential locality in NVRAM. Experimental results compliant to the Graph500 benchmark on a single compute node with arrays of NAND flash-based SSDs show that NETALX can achieve 4.14 Giga TEPS (Traversed Edges Per Second) for a graph with 2^{31} vertices and 2^{35} edges, whose size is 4 times larger than the size of graphs that the machine can accommodate only using DRAM, with only 14.99% performance degradation. We also demonstrate that NETALX can achieve a power efficiency of 11.8 Mega TEPS/W (Traversed Edges Per Second / Watt).

Second, for large-scale dynamic graph analytics, we propose DegAwareRHH, a high performance dynamic graph data store, which leverages a linear probing open addressing compact hash table that exhibits high spatial and sequential locality to increase graph update performance keeping graph analytics performance. To extend DegAwareRHH to distributed-memory platforms, we adopt an asynchronous communication framework aiming for localizing remote communication into the area where need to be updated. We demonstrate that DegAwareRHH is 212.2 times faster than a state-of-the-art shared-memory streaming graph processing framework on a single compute node to update a graph with 1 billion edge insertion requests and 54 million edge deletion requests. DegAwareRHH achieves a processing rate of over 1.8 billion edge insertion requests per second at 192 compute nodes on a massive-scale real graph that has 128 billion edges. We also show that DegAwareRHH can accelerate the performance of a large-scale dynamic graph colouring algorithm and achieve high performance on out-of-core graph update workloads including future NVRAM devices.

This thesis presents several contributions towards high performance data stores for large-scale graph analytics, in terms of locality awareness, on HPC platforms, including next generation supercomputers which will have locally-attached NVRAM, such as Tsubame 3.0 at Tokyo Institute of Technology and Sierra at Lawrence Livermore National Laboratory.

Acknowledgements

I would like to express my special appreciation and thanks to my advisor Prof. Satoshi Matsuoka. I would like to thank you for encouraging my research and for allowing me to grow as a research scientist. Your advice on both research as well as on my career have been priceless. I owe sincere and earnest thankfulness to Dr. Hitoshi Sato, Mr. Yuichiro Yasui, Prof. Katsuki Fujisawa, and the members at Tokyo Institute of Technology, especially the members at Matsuoka laboratory, for their help and support for research activities and for providing me with an excellent atmosphere for doing research. My sincere thanks also go to Dr. Roger Pearce, Dr. Brian Van Essen and Dr. Maya Gokhale, who provided me an opportunity to join their team as an intern at Lawrence Livermore National Laboratory, and gave me access to the laboratory and research facilities. Without their precious support, I havn't conducted this research. Words cannot express how grateful I am to them. I would like to express my greatest gratitude to the people who provided essential components to perform my research work and conducted experiments for this work. Mr. Scott Sallinen at University of British Columbia who developed the dynamic graph colouring algorithm and performed experiments; Mr. Suraj Poudel at University of Alabama at Birmingham who collected the Wikipedia graph dataset; Dr. Manu Shantharam, Dr. Pietro Cicotti and Dr. Laura Carrington at San Diego Supercomputer Center who conducted experiments on the NVRAM emulator. I would like to thank Japan Society for the Promotion of Science (JSPS) and Japan Science and Technology Agency (JST), which gave me the great opportunity to study at Tokyo Institute of Technology. Finally, I would also like to show a special thanks to my family. Without their support, this thesis could never be completed.

Keita Iwabuchi, February 2017

Publications

Conference (refereed)

[1] Keita Iwabuchi, Hitoshi Sato, Yuichiro Yasui, Katsuki Fujisawa, and Satoshi Matsuoka, “NVM-based Hybrid BFS with Memory Efficient Data Structure”, In Proceedings of the 2014 IEEE International Conference on Big Data (IEEE BigData 2014), October 2014.

[2] Scott Sallinen, Keita Iwabuchi, Suraj Poudel, Maya Gokhale, Matei Ripeanu, and Roger Pearce, “Graph Colouring as a Challenge Problem for Dynamic Graph Processing on Distributed Systems”, In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC16), November 2016.

Workshop (refereed)

[3] Keita Iwabuchi, Scott Sallinen, Roger Pearce, Brian Van Essen, Maya Gokhale, and Satoshi Matsuoka, “Towards a Distributed Large-Scale Dynamic Graph Data Store”, In Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (GABB’2016), May 2016.

[4] Keita Iwabuchi, Hitoshi Sato, Ryo Mizote, Yuichiro Yasui, Katsuki Fujisawa, and Satoshi Matsuoka, “Hybrid BFS Approach Using Semi-external Memory”, In Proceedings of the 2014 IEEE International Parallel Distributed Processing Symposium Workshops (HPDIC2014), May 2014.

Poster (refereed)

[5] Keita Iwabuchi, Roger Pearce, Brian Van Essen, Maya Gokhale, and Satoshi Matsuoka, "Design of a NVRAM Specialized Degree Aware Dynamic Graph Data Structure", International Conference on High Performance Computing, Networking, Storage and Analysis (SC15) (Poster), November 2015.

[6] Keita Iwabuchi, Hitoshi Sato, Ryo Mizote, Yuichiro Yasui, and Katsuki Fujisawa, "Performance Analysis of Hybrid BFS Approach Using Semi-External Memory", International Conference on High Performance Computing, Networking, Storage and Analysis (SC13) (Poster), November 2013.

Domestic Workshop (non-refereed)

[7] Iwabuchi Keita, Sallinen Scott, Pearce Roger, Van Essen Brian, Gokhale Maya, and Matsuoka Satoshi, "Towards a Distributed Large-Scale Dynamic Graph Data Store", IPSJ SIG Technical Report 2016-HPC-153, February 2016.

Contents

Abstract **i**

Acknowledgements **iii**

Publications **iv**

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Statement 2

1.3 Approaches and Contributions 3

1.4 Thesis Outline 5

2 Background **7**

2.1 Graph 7

2.1.1 Graph Theory 7

2.1.2 Graphs in the Real World 7

2.1.3 Small World and Scale-free 8

2.2 Non-volatile Random-Access Memory (NVRAM) 10

2.3 Graph Data Structure 11

2.3.1 Static Graph Data Structure 12
2.3.2 Dynamic Graph Data Structure 13
2.4 Graph Analytics Workload 14
2.4.1 Graph Traversal Algorithm 15
2.5 Graph500 16
2.5.1 Graph Generator 17

3 Out-of-core Static Graph Data Store 19

3.1 Introduction 19
3.2 Preliminaries 20
3.2.1 Hybrid BFS Algorithm 20
3.3 Out-of-core Hybrid BFS 25
3.3.1 NUMA-Optimized Hybrid BFS Implementation 25
3.3.2 Out-of-core Hybrid BFS with Dual Graph Model 26
3.3.3 Problems for Scaling Graphs Using NVRAM 26
3.4 Out-of-core Hybrid BFS with Memory Efficient Data Structure 27
3.4.1 Hybrid BFS with Single Graph Model 27
3.4.2 Analysis of Data Accesses in Bottom-up Approach 28
3.4.3 Out-of-core Hybrid BFS with Single Graph Model 29
3.4.4 Implementation Details 30
3.5 Experiments 36
3.5.1 Experimental Setup 36
3.5.2 Memory Consumption 38
3.5.3 BFS Performance for Large-scale Graphs 38

3.5.4 BFS Performance on Reduced Memory Configuration 40

3.5.5 BFS Performance Changing Edge_factor Parameters 41

3.5.6 Power Efficiency Using NVRAM 43

3.6 Related Work 44

3.7 Discussion: Performance Evaluation on Normal (not scale-free) Graph . 46

3.8 Summary 47

4 Large-Scale Dynamic Graph Data Store 49

4.1 Preliminary 50

4.1.1 Dynamic Graph Analytics 50

4.1.2 Design Principle 51

4.1.3 Underling Data Structure 51

4.2 Robin Hood hashing (RHH) 52

4.2.1 Insert 53

4.2.2 Delete 53

4.2.3 Search 54

4.3 DegAwareRHH 54

4.3.1 Overview 55

4.3.2 Data Layout 55

4.3.3 Programming API 56

4.3.4 Insert Algorithm 57

4.3.5 Delete Algorithm 58

4.3.6 Optimization 60

4.4 Extending DegAwareRHH for Distributed-Memory 61

4.4.1 An Asynchronous Distributed-Memory Communication Framework 61

4.4.2 DegAwareRHH on the Asynchronous Visitor Queue 62

4.5 Experimental Setup 63

4.5.1 Details of Implementation of DegAwareRHH 63

4.5.2 Implementations for Comparison 64

4.5.3 Datasets 65

4.5.4 Machine Configurations 67

4.6 Experiment 1: Large-scale Dynamic Graph Construction 68

4.6.1 Experiment Method 68

4.6.2 Single Process Experiments 68

4.6.3 Single Node Experiments (vs STINGER) 70

4.6.4 Multi-Node Experiments 73

4.6.5 Realistic Workload 78

4.7 Experiment 2: Performance Evaluations on Graph Algorithms 79

4.7.1 Experiment Method 80

4.7.2 Results 80

4.8 Experiment 3: Dynamic Graph-colouring at Large-scale 84

4.8.1 Graph Colouring 84

4.8.2 Distributed Dynamic Graph Colouring Algorithm 85

4.9 Experiment 4: Out-of-core Dynamic Graph Construction 89

4.10 Experiment 5: Dynamic Graph Construction in Future NVRAM Technologies 90

4.10.1 Emulator 90

4.10.2 Experimental Setup 91
4.10.3 Result 92
4.11 Related work 93
4.12 Discussion: Order of Edge Stream 94
4.13 Summary and Future Work 95
4.13.1 Summary 95
4.13.2 Future Work 96

5 Conclusion and Future Work 97

5.1 Conclusion 97
5.2 Future Work 99

Bibliography 100

List of Figures

2.1 An Example of a Graph 8

2.2 THE INTERNET 2003; Visualization of the routing paths of the Internet 9

2.3 An Example of a Power-law Distribution ($\gamma = 2.45$) 10

2.4 Adjacency Matrix 12

2.5 CSR Graph Data Structure 13

2.6 Adjacency-list Data Structure 14

2.7 An Example of a BFS 16

2.7a Given Graph 16

2.7b Result of a BFS (BFS tree) 16

3.1 Outline of Top-down Approach 21

3.2 The Breakdown of Traversed Edges by Top-down Approach 22

3.3 Breakdown of Edges in the Frontier for a Sample Search on kron27 . . . 23

3.4 Outline of Bottom-up Approach 24

3.5 Analysis of Data Accesses in Bottom-up Approach 30

3.5a Number of Source Vertex that Scans m Edges 30

3.5b Number of Traversed Edges by Source Vertex and Completion Ratio 30

3.6 Data Structures of Out-of-core Hybrid BFS with Single Graph Model . 31
3.7 Out-of-core Graph Construction 36
3.8 Overview of EBD-I/O device 37
3.9 Comparison of Memory Consumption 38
3.10 BFS Performance in TEPS 39
3.11 BFS Performance and Memory Consumption 41
3.12 Average Execution Times in Both Approaches 42
3.13 Number of Accesses to NVRAM in Bottom-up Approach 42
3.14 BFS Performance in TEPS with changing edge_factor 43
3.14a DRAM-only 43
3.14b DRAM+NVRAM 43
3.15 Average Power Consumption 45
3.16 Power Efficiency Metric in MTEPS/W 45
3.17 The Degree Distribution of a RMAT-ER graph with $edge_factor = 16$. 46
3.18 Out-of-core BFS Performance on a RMAT-ER Graph 47

4.1 Edge insert into Robin Hood hashing table 54
4.2 Overview of Degree Aware Dynamic Graph Data Structure (DegAwareRHH) 57
4.3 Degree Aware Edge Insert Algorithm 58
4.4 Degree Aware Edge Delete Algorithm 59
4.5 Distributed Dynamic Graph Update over the Visitor Queue Framework 63
4.6 Single Process Unique Edge Insertion and Deletion 69
4.7 The Variations of Average Probe Distance When Constructing a Graph 71

4.7a Low-Degree Table 71

4.7b Middle-High-Degree Table 71

4.8 The Distribution of the Average Probe Distances of the Edge-Chunks . 72

4.9 Speed Up for Unique Edge Insertion and Deletion Against STINGER . 72

4.10 Single-node Unique Edge Insertion and Deletion 73

4.11 Multi-node Non-unique Edge Updates 75

4.11a Insertion only 75

4.11b 5% of additional deletion 75

4.12 Multi-node Unique Edge updates 76

4.12a Insertion only 76

4.12b 5% of additional deletion 76

4.13 Muti-node Unique Edge Insertion on Real Graphs 77

4.13a Twitter 77

4.13b SK2005 77

4.14 Multi-node Unique Edge Insertion on Webgraph2012 78

4.15 Unique Edge Insertion on the Wikipedia Graph 79

4.16 Single Process BFS Algorithm 81

4.17 Single Process PageRank Algorithm 81

4.18 The Performance of Single Process BFS 82

4.19 The Performance of Single Process PageRank 83

4.20 An Example of the Dynamic Graph Colouring Algorithm 86

4.21 Run-time for Dynamic Graph Colouring on the Real-world Graphs . . . 88

4.21a Twitter 88

4.21b SK2500 88

4.22 Run-time for Dynamic Graph Colouring on the Wikipedia Graph 88

4.23 Out-of-core Unique Edge Insertion 89

4.24 Architecture of the Software Emulator Platform (SEP) 91

4.25 Out-of-core Unique Edge Insertion Varying Read Latency From 100 to 350 ns 92

List of Tables

2.1 Comparison of Memory Technologies; data from [72] 11

3.1 The number of traversed edges by Top-down (m_F), Bottom-up (m_F) and Hybrid (*oracle*) in BFS on a kronecker graph 25

4.1 Candidates for Underling Data Structure 52

4.2 The Size in Bytes of Each Element in Tables in DegAwareRHH 64

4.3 Static Graph Datasets Used in Experiments 67

4.4 Properties of Wikipedia Dataset 67

Chapter 1

Introduction

1.1 Motivation

Recently, the amount of data in the world is growing rapidly. Indeed, according to a report publish by Cisco, annual global IP traffic will exceed a zettabyte (10^{21}) by the end of 2016 and will reach 2.3 ZB per year by 2020 [11]. It is anticipated that huge amount of data will flow into large computer systems due to the emergence of the huge volume of open data and development of Internet of Things (IoT). This phenomenon, called Big Data, brings us many challenges attributed to not only volume but also to the emergence of a new paradigm; that is, analyzing the data to discover knowledge, to understand behaviors, and to mine for patterns [46]. Given this paradigm, computation model is shifting to complex data access patterns and requires management of large volume of data in contract to the traditional computation pattern that is compute-bound. Because the bottleneck of such applications is I/O operation, they are described as data-intensive computing.

At the same time, demands for large-scale graph analytics has risen as an important kernel for HPC applications in various domains, such as World Wide Web (WWW) and social network analysis, network security, artificial intelligence and genomic analysis. Due to the explosion of data in the recent years, the size of graphs appear in the real world has been rapidly increasing. For example, Facebook manages 1.39 billion active users as of 2014, and their social network graph has more than 400 billion edges [33]; the indexed web contains at least 4.83 billion pages as of 2016 [24, 9]. However, large graph analytics is one of the representative of data-intensive problems and there

are many challenges. In fact, rapidly increasing numbers of these applications cause significant attractions to the bi-annual Graph500 list [3], which ranks supercomputers based on their performance in executing large-scale graph problems as an example of data-intensive workloads.

In addition, for exascale computing, providing sufficient main memory capacity is one of the biggest challenges due to the cost and high power consumption of DRAM [54]. For instance, an exascale supercomputer is required to solve science problems 10x–50x faster than current top supercomputers without increasing power consumption, in a power envelope of 20-30 megawatts [2]. Due to this constraint, interest in Non-volatile random-access memory (NVRAM) has risen. Although NVRAM has lower throughput and higher latency compared with DRAM, node-local NVRAM has found its way into HPC platforms and has enabled the possibility to extend main-memory capacity without extremely high cost and power consumption to cope with such explosion of data. Currently, the major technology used to implement NVRAM is NAND flash, and there are multiple emerging technologies to implement NVRAM, including Phase Change Memory (PCM), STT-MRAM, and resistive RAM (ReRAM).

1.2 Problem Statement

Large-scale graph analytics often presents challenging data-intensive workloads, due to unstructured and random memory access patterns. Such memory access patterns result in low data locality and low memory utilization; a naive implementation of a large-scale graph store will result in significant performance degradation. Therefore, designing data stores for graph analytics is an extremely important factor for enabling high-performance graph analytics. Nevertheless, the study of data stores for large-scale graph analytics is still an unexplored area.

Significant research into the analysis of challenging large-scale graphs that are static (graph topology remains fixed) has resulted in remarkable improvements in the processing capabilities of HPC systems [94, 26, 31]; much of this work can be attributed to the Graph500. However, little study has been done to investigate how to utilize NVRAM and how it impacts the performance of large-scale graph analytics. More specifically, an efficient implementation based on detailed analysis of access patterns of unstructured graph kernels on a system that utilize a mixture of DRAM and NVM

devices has not been well investigated in literature.

In addition, in many real-world graph applications, the structure of the graph changes dynamically over time and may require real-time analysis. Repeatedly re-analyzing large-scale graphs often cannot keep up with performance requirements; thus, it is required to develop the data store and infrastructure management necessary to support dynamic graph analytics at large scale, on distributed HPC platforms, including next generation supercomputers which have locally-attached NVRAM. However, research into data stores for dynamic graph analytics at large scale is at the moment immature compared with the static scenario. While the need for such capabilities is strong, studies for storing large-scale dynamic graphs are scarce even without taking NVRAM into account.

1.3 Approaches and Contributions

To address the above issues, we explore techniques and design of large-scale graph data stores from the perspective of static and dynamic graph analytics. Although graph analytics generally suffers from lack of data locality, designing graph data store based on the characterization of graph analytics workloads has potentials to enable high-performance graph analytics [21]. This thesis presents several contributions towards high performance data store for large-scale graph analytics on HPC platforms, including next generation supercomputers which have locally-attached NVRAM.

Out-of-core Static Graph Data Store Our study starts by targeting static graphs. We first propose a graph data offloading technique using NVRAM that augments the Hybrid BFS (Breadth-First Search) algorithm [20], considered to be one of the fastest BFS algorithms for scale-free and small-world graphs and widely used in the Graph500 [3] benchmark especially amoung top performance implementations of the list [30, 50]. Our technique arranges graph data to improve data locality (reduce the number of accesses to NVRAM) and sequential locality in NVRAM. Specifically, deploys frequently-accessed graph data with fine-grained I/O size into DRAM memory space based on detailed analysis of memory access patterns of the algorithm and a property of NVRAM, i.e., fine-grained I/O causes huge overhead.

The key contributions include:

- Using the graph data offloading technique, we develop NETALX, an out-of-core Hybrid BFS implementation based on a highly NUMA-optimized in-core Hybrid BFS implementation called NETAL (NETwork Analysis Library) [92, 93], one of the fastest single-node implementation on the Graph500 list [3].
- We demonstrate that NETALX achieves extremely high-performance BFS execution for large-scale graphs whose size exceed the capacity of DRAM on the machine. Experimental results on Kronecker (synthetic) graphs compliant with the Graph500 benchmark on a 2-way Intel Xeon E5-2690 machine with 256 GB of DRAM and arrays of NAND flash SSDs in RAID 0 show that NETALX can achieve 4.14 Giga TEPS (Traversed Edges Per Second) for a SCALE31 graph problem with 2^{31} vertices and 2^{35} undirected edges, whose size is 4 times larger than the size of graphs that the machine can accommodate only using DRAM, with only 14.99 % performance degradation.
- We also show that NETALX can achieve a power efficiency of 11.8 Mega TEPS/Watt on the SCALE 31 Kronecker graph. In other words, NETALX can run BFS with the same power consumption and power efficiency on a 4 times larger graph compared with the case where only DRAM is used. Based on our implementation and with further optimizations, we achieved the 3rd and 4th position of the Green Graph500 list (2014 June) in the Big Data category.

Large-scale Dynamic Graph Store We aim to develop a large-scale dynamic graph data store that can scale beyond trillions of edges, and especially it is targeting incremental dynamic graph analytics frameworks on distributed HPC platforms, including next generation supercomputers which have locally-attached NVRAM.

We propose a large-scale dynamic graph data store (*D*egAwareRHH), which leverages a linear probing and open addressing compact hash table that exhibits high space and sequential locality, in order to minimize 1) the overhead of reading adjacent edges of a vertex; 2) cache misses and page misses. In addition, DegAwareRHH is *degree aware*, and uses separated compact data structures for low-degree vertices to reduce their storage and search overheads on NVRAM. We extend DegAwareRHH for distributed-memory platforms using an *a*synchronous MPI communication framework [70, 71] aiming for localizing remote communication in the area where graph update occurred.

Summary of our contributions include:

- We demonstrate that DegAwareRHH can process 1 billion edge insertion requests and 5% of additional edge deletion requests 212.2 times faster than STINGER[39], a state-of-the-art shared-memory streaming graph processing framework, when both implementations use 24 threads/processes.
- We present scaling studies of constructing large-scale real-world graphs including a massive-scale hyperlink graph which has 128 billion edge insertion requests, and show DegAwareRHH processes over 1.8 billion edge insertion requests per second at 192 compute nodes.
- We also show DegAwareRHH can accelerate the performance of a massive-scale dynamic graph colouring algorithm.
- Finally, we demonstrate that DegAwareRHH also achieves high performance on out-of-core workloads including future NVRAM devices by using a NVRAM emulator.

1.4 Thesis Outline

The dissertation is divided into five chapters, and is organized as follows:

Chapter 2: Background

We first introduce basic knowledge and technologies of graph theory, NVRAM, and major graph data structures followed by challenges for high performance graph analytics at large-scale. We also list two major graph algorithms: breadth-first search (BFS) and PageRank.

Chapter 3: Out-of-core Static Graph Data Store

We propose our graph data offloading technique using NVRAM that augments the Hybrid BFS (Breadth-First Search) algorithm [20] and describe the implementation of NETALX, our extremely-fast out-of-core Hybrid BFS implementation. We demonstrate that NETALX achieves extremely high-performance and power-efficient BFS execution for large-scale static graphs whose size exceed the capacity of DRAM on the machine.

Chapter 4: Large-scale Dynamic Graph Data Store

In this chapter, we propose DegAwareRHH, a novel high performance dynamic graph data store. We first explore and describe design of DegAwareRHH in terms of how to store a graph into local memory space efficiently and extend for distributed memory. Then, we experimentally evaluate the performance of DegAwareRHH in terms of principal graph analytics workloads such as graph traversal, accessing property data, and graph update on in-core workloads and out-of-core workloads including future NVRAM using a NVRAM emulator.

Chapter 5: Conclusion and Future Work

Finally, we summarize the contributions made by our work and discuss possible directions for future research.

Chapter 2

Background

2.1 Graph

2.1.1 Graph Theory

A graph $G(V, E)$ consists of a set of vertices or nodes $V(G) = \{v_1, v_2, v_3, ..., v_n\}$ and a set of edges $E(G) = \{e_1, e_2, e_3, ..., e_m\}$. An edge is a 2-element subset of V. An example of a graph comprised of a set of vertices $V = \{v_1, v_2, v_3, v_4, v_5\}$ and a set of edges $E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$ is illustrated in Figure 2.1. Each vertex and edge can have property data. The degree of a vertex is the number of edges the vertex has; for instance, the degree of vertex v_1 is 3. A graph of which edges have no orientation is said undirected graph; an edge (x, y) is identical to an edge (y, x). On the other hand, a graph of which edges have orientations is said directed graph.

2.1.2 Graphs in the Real World

In the real world, many things can be represented as graphs. Examples of real-world graphs are as follows:

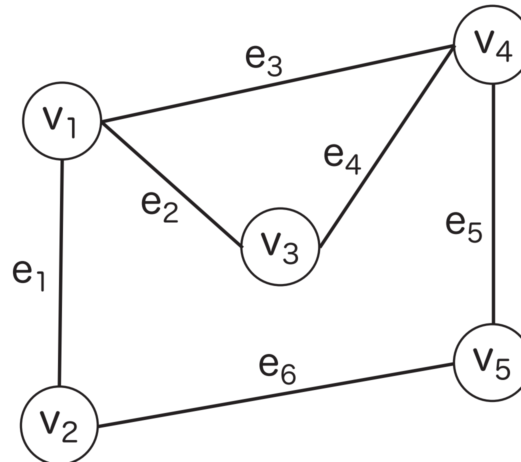


Figure 2.1: An Example of a Graph

The Internet / World Wide Web Figure 2.2 is a map of the Internet as of 2003 crawled and visualized by Barrett Lyon / The Opte Project [6]; each vertex represents a host/router in the Internet and each edge represents that there is a communication between two vertices. For World Wide Web, the indexed web contains at least 4.83 billion pages as of 2016 [24, 9]. As for dynamic changes of graphs, the peak inbound and outbound unicast packets per second in the Seattle Internet Exchange reach 140 million and 150 million, respectively [7].

Social Networking Service (SNS) Facebook manages 1.39 billion active users as of 2014, and their social network graph has more than 400 billion edges [33]. Twitter has 41.7 million users, and the users are connected to each other by 1.4 billion follow links as of 2009 [52].

Brain Network In neuroscience, a brain network can be represented as a graph. For instance, the human brain has approximately 100 billion neurons and 100 trillion synapses [4].

2.1.3 Small World and Scale-free

Characteristics of real world graphs have been studied since the late of the 1950s [36, 38, 43]. Here we describe two important characteristics: small world and scale-free.

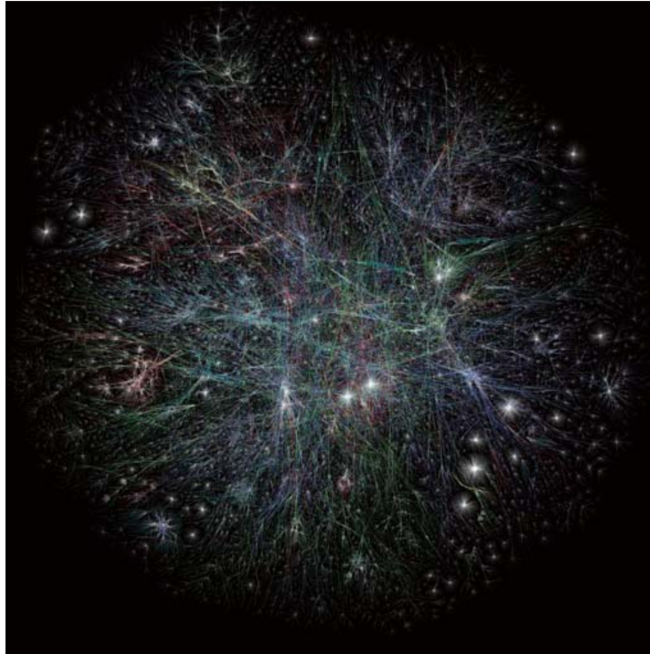


Figure 2.2: Figure from [6]; THE INTERNET 2003; Visualization of the routing paths of the Internet

2.1.3.1 Small World

In graph theory, the distance L of vertex v_j and vertex v_i is the minimum number of edges between them. Small-world network is a network of which typical distance L between two randomly chosen vertices is small even large number of vertices the graph has, that is:

$$L \propto \log aN \tag{2.1}$$

where a is a parameter.

Small-world properties are found in many real-world graphs [63, 89, 15, 12]. For instance, Facebook reported that each user is connected to every other user by 3.57 users on average (there is 1.59 billions of active users on Facebook as of 2016) [82].

2.1.3.2 Scale-free Graph

It has been reported that many real-world graphs can be classified as scale-free, where the distribution of vertex degrees follows a scale-free power-law distribution [14,

18, 15]. We can write the degree distribution as:

$$p(k) \propto k^{-\gamma} \tag{2.2}$$

where k is a degree, γ is a parameter whose value is typically in the range $2.0 \leq \gamma \leq 3.4$ [13, 34, 83]. A degree distribution with $\gamma = 2.45$, which can be seen in the structure of the World Wide Web, is shown in Figure 2.3. A power-law vertex degree distribution means that the majority of vertices have a low-degree, while a select few have a very large degree, and its degree distribution follows a power-law pattern.

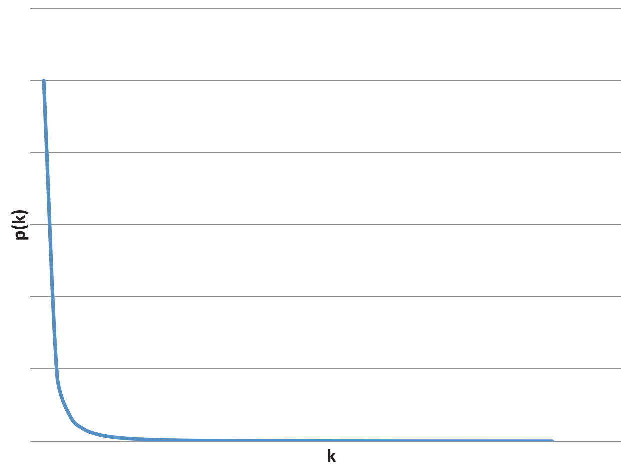


Figure 2.3: An Example of a Power-law Distribution ($\gamma = 2.45$)

Recent work regarding large-scale graph processing has focused on optimizations related to the challenge of high-degree vertices which cause load imbalance for parallel computations [41, 70, 71]. In this work, we identify low-degree vertices which are particularly challenging for graph data stores in NVRAM.

2.2 Non-volatile Random-Access Memory (NVRAM)

In the next generation of supercomputers, providing sufficient main memory capacity is one of the biggest challenges due to the high power consumption and cost of dynamic random-access memory (DRAM) [54]. Therefore, NVRAM has attracted considerable attention [72] to extend main memory capacity. Nowadays, NAND flash

Table 2.1: Comparison of Memory Technologies; data from [72]

Parameter	DRAM	NAND flash	SCM (PCM, ReRAM)
Density	1×	4×	2×–4×
Read Latency	1×	2^8×	4×

technology is mainly used to implement actual NVRAM products. NVRAM will greatly expand the possibility of processing extremely large-scale graphs that exceed the DRAM capacity of the nodes; however, graph construction cost using NVRAM is extremely high compared with DRAM – a naive data structure implementation would cause significant performance degradation due to unstructured memory accesses.

In addition to NAND flash, there are multiple emerging technologies to implement NVRAM that are expected provide better performance than NAND flash including Phase Change Memory (PCM) and resistive RAM (ReRAM). A comparison of such emerging memory technologies is shown in Table2.1. Emerging such new NVRAM technologies, PCM and ReRAM, are expected to provide DRAM like latency as persistent memory [16] and to provide high performances on graph analytics [59]. Intel and Micron recently announced a new NVRAM product called 3D XPoint technology memory [1] that is designed to play a role between DRAM and NAND flash in terms of performance and cost.

2.3 Graph Data Structure

Graph processing is a highly data-intensive problem, thus improving the data locality of graph data structures is an essential optimization [65]. Over the years, many data structure models have been studied. In this section, we discuss classic graph data structure models for static graphs and dynamic graphs and their advantages and disadvantages. In the following context, we consider a graph G which has n vertices and m edges.

Actually, these days, leveraged by computation power of Graphics Processing Unit (GPU), many studies have been conducted that perform matrix multiplication for graph analytics, for instance [44, 47, 62, 88]; however, our study only targets CPU based graph analytics computation models.

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix}$$

Figure 2.4: Adjacency Matrix

2.3.1 Static Graph Data Structure

2.3.1.1 Adjacency-matrix

The simplest data structure model is an *adjacency-matrix* A. The adjacency-matrix of the graph G is an $n \times n$ matrix (Figure 2.4). If there is an edge from v_i to v_j, an entry a_{ij} holds a positive value or property data of the edge. The adjacency-matrix consumes $O(n^2)$ memory, which is a huge disadvantage when processing sparse graph.

2.3.1.2 Compressed Sparse Row (CSR)

The CSR data structure is a de facto graph data structure widely used in many static graph processing implementations. The CSR data structure consists of two array structures called *index* array and *edge* array. The *index* array holds indices to the *edge* array, and the *edge* array holds adjacent vertices' ID. More specifically, each index of the *index* array represents a source vertex's ID, and the corresponding element in the *index* array refers to an index of the *edge* array. The range of a vertex's edges in the *edge* array is from $index[v_i]$ to $index[v_i + 1]$. The memory size of the CSR data structure is $O(n+1)$ for the *index* array and $O(m)$ for the *edge* array. An overview of the CSR data structure is illustrated in Figure 2.5. The CSR data structure can provide high data locality and efficient memory usage owing to its packed array structure. However, because of packing, the CSR data structure is not well suited for storing dynamic graphs. In general, even when adding or deleting a single edge or vertex, the CSR data structure requires updates to the entire space, causing large data movement.

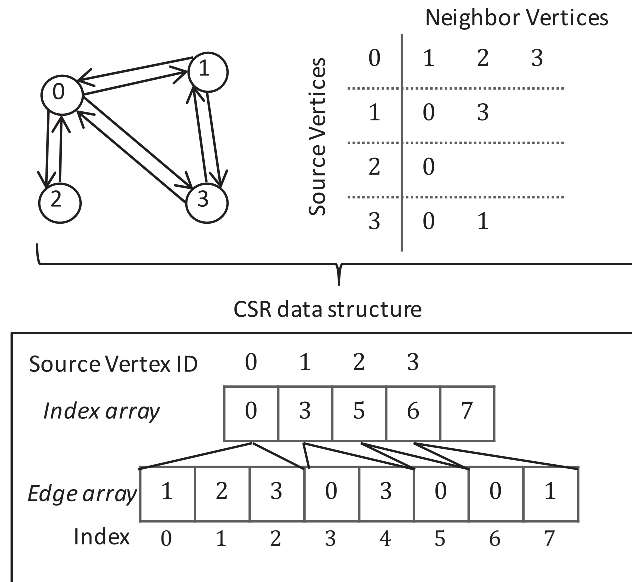


Figure 2.5: CSR Graph Data Structure

2.3.2 Dynamic Graph Data Structure

2.3.2.1 Key-Value

The Key-Value Store (KVS) model, in other words a map or dictionary data structure, manages an element as pair of a *key* and *value*. In a simple KVS graph model, an edge table uses the source and target edge pair as the *key* and the edge property as the *value*. To store vertex data, for example, the KVS holds a *vertex table* consisting of pair of a vertex and vertex property data. However, while such a simple KVS model can insert and delete a vertex and edge efficiently at large-scale, there is no consideration of graph topologies. Therefore, it is difficult to obtain locality benefits among edges adjacent to a vertex, which is a highly important factor to determine the performance of graph analytics.

2.3.2.2 Adjacency-list

The basic components of an adjacency-list are a *vertex-table* and *edge-list* (Figure 2.6). A vertex-table holds a set of all vertices of a graph. Each element of a vertex-table consists of a pointer to an edge-list and also vertex property data if

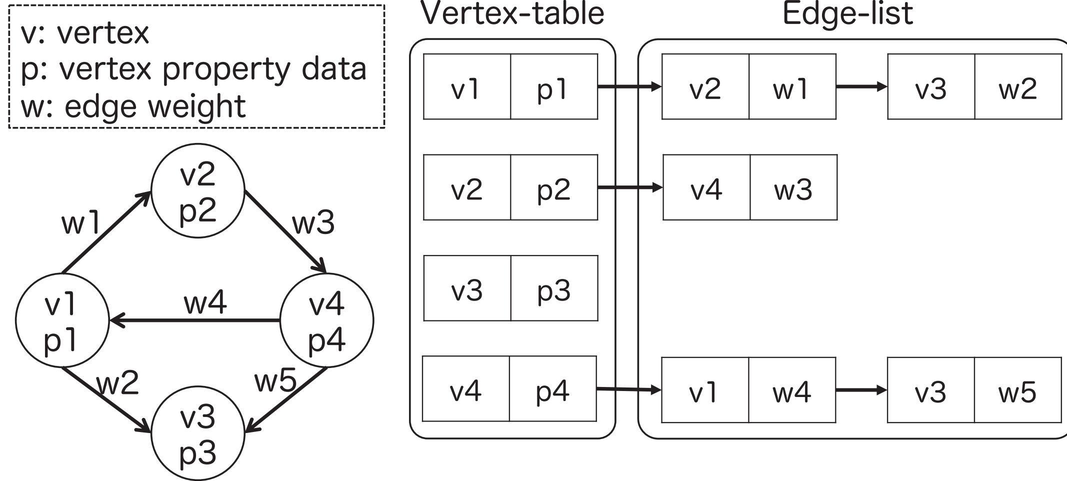


Figure 2.6: Adjacency-list Data Structure

needed. An edge-list holds the list of neighbors of a vertex with edge property data also if needed. The main advantage of an adjacency-list model is that it can provide high data locality among out-going edges (or in-coming, depends on a configuration) of each vertex. There are many variations on this model, each with its advantages and disadvantages. For a vertex-table, typically a tree or hash (key-value) data structure is used, and for an edge-list, a single vector or linked-list data structure is used.

The tree data structure is widely used in database systems; however, a tree data structure potentially causes random memory accesses for each operation (such as, *insert, find and delete*), and if we store the data structure into external-memory layer, it leads overhead. Even though the time complexity of a tree data structure is $O(log(n))$, it is too costly for large-scale graphs especially stored in NVRAM; therefore, we use a near-constant time hash table.

2.4 Graph Analytics Workload

In graph analytics, its principal computation types are can be classified into three computation types [65]. The first type is graph updation, that is, insert or delete vertices or edges dynamically. The second type is accessing property data of a graph, that is, get/set property data of a vertex/edge or get a degree of a vertex. The last type is graph traversal; we will describe two actual algorithms in the following section.

2.4.1 Graph Traversal Algorithm

In this section, we describe two basic and important graph algorithms: Breadth-First Search (BFS) and PageRank. We use the algorithms to evaluate our graph store.

2.4.1.1 Breadth-First Search (BFS)

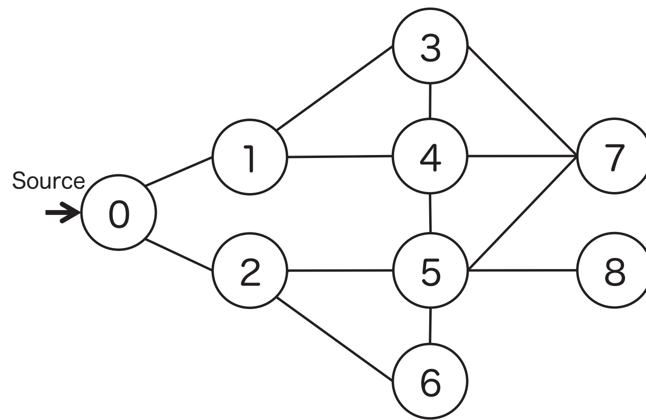
Breadth-First Search (BFS) is usually considered as a representative graph algorithm and actually used in many real-world use cases [65]. BFS is also an important building block in many other graph algorithms. A BFS explores all the vertices in a graph that can be reached from v_s, a source (or root) vertex of the traversal. The result of a BFS starting from vertex 0 is shown in Figure 2.7. After performing the BFS, the distance and shortest path to each vertex from the source vertex can be obtained, e.g., vertex 7 is reachable from vertex 0 by taiking the path $0 \rightarrow 1 \rightarrow 3 \rightarrow 7$ with 3 ditance. The bottom figure in Figure 2.7 is called BFS tree. Note that it depends on an actual BFS algorithm that which paths (edges) are chosen to construct a BFS tree – for instance, edge $4, 7$ can be chosen instead of edge $3, 7$ to construct the BFS tree.

2.4.1.2 PageRank

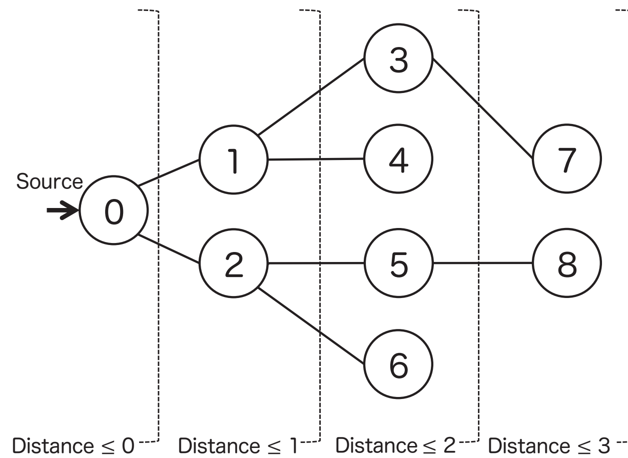
PageRank [67] is also considered as a representative graph algorithm. PageRank is a way of determine website pages' relevance or importance. The PageRank (PR) value of a page p can be expressed with a simplified model as:

$$PR(p) = \sum_{q \in p_{in}} \frac{PR(q)}{m_q} \tag{2.3}$$

where p_{in} is a set of all pages that have links to page p and m_q is the number of out-going links page q has.



(a) Given Graph



(b) Result of a BFS (BFS tree)

Figure 2.7: An Example of a BFS

2.5 Graph500

The Graph500 list [3] ranks computers by executing a set of benchmarks for large-scale graph problems. The Graph500 list is announced at two international conferences in high performance computing field: International Supercomputing Conference (ISC) and Supercomputing Conference (SC). In contrast to the Top500 list [8], which is known as a list that ranks computers by executing the Linpack benchmark as an instance of compute-intensive workloads, Graph500 adopts graph processing as an instance of data-intensive workloads. Specifically, the current benchmark in Graph500 measures the time for performing BFS to a Kronecker graph [56, 55], which models a real-world graph, from randomly selected 64 start points.

The detailed instructions of the Graph500 benchmark are described as follows:

Step1 Edge List Generation

First, the benchmark generates an edge list of an undirected graph with $n(=2^{SCALE})$ vertices and $m(= n \cdot edge_factor)$ edges.

Step 2 Graph Construction

Second, the benchmark constructs a suitable data structure, such as CSR (Compressed Sparse Row) graph format, for performing BFS from the generated edge list. The execution time of this step is not used to rank the list.

Step 3 BFS

Then, the benchmark run a BFS to the constructed data structure to create a BFS tree. Graph500 employs TEPS (Traversed Edges Per Second) as a performance metric. Thus, the elapsed time of a BFS execution and the total number of edges the graph has used to compute the performance of the benchmark.

Step 4 Validation

Finally, the benchmark verifies the results of the BFS tree. This step is untimed.

The benchmark iterates Step 3 and Step 4 64 times from randomly selected start points (vertices), and the median value of the results is used as the score of the benchmark.

2.5.1 Graph Generator

To perform the benchmark on large-scale graph datasets that have the same properties can be found in real-world graphs, scale-free and small world, Graph 500 generates synthetic graphs using the Recursive MATrix (R-MAT) graph generator model [28] or Kronecker product [56, 55].

2.5.1.1 R-MAT Graph Generation Model

Let P be an initial 2×2 matrix where the sum of all elements is 1.0. In R-MAT graph generation model, subdivide the number of edges required to generate recursively into 4 spaces with the probabilities in P until the size of each divided space became 1.

2.5.1.2 Kronecker Graph Generation Model

Let A be a $n \times m$ matrix and B be a $k \times l$ matrix. Kronecker product of the matrices $C = A \bigotimes B$ is given by

$$C = A \bigotimes B = \begin{bmatrix} a_{1,1}B & a_{1,2}B & \cdots & a_{1,m}B \\ a_{2,1}B & a_{2,2}B & \cdots & a_{2,m}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1}B & a_{n,2}B & \cdots & a_{n,m}B \end{bmatrix} \tag{2.4}$$

where C is a $(nk) \times (ml)$ matrix.

To generate any size of graphs, the Kronecker product is iteratively applied. Let G_1 be an initial adjacency matrix of a graph that has N_1 nodes and E_1 edges.

$$G_k = \underbrace{G_1 \bigotimes G_1 \bigotimes \cdots G_1}_{k\ times} \tag{2.5}$$

As iterates the Kronecker product, the size of the graph increases exponentially. Thus, G_k has N_1^k veritces and E_1^k edges. Because Kronecker Product can generalize R-MAT graph model, by setting G_1 to a 2×2 matrix where the sum of all elements is 1.0 like used in R-MAT model, the Kronecker graph generator model is equal to R-MAT model.

Chapter 3

Out-of-core Static Graph Data Store

3.1 Introduction

Large-scale graph processing in various application domains such as health care, systems biology, social networks, business intelligence, and electric power grids, etc., is considered an important kernel for HPC applications. In fact, rapidly increasing numbers of these applications cause significant attractions to the Graph500 list [3], which ranks supercomputers by executing large-scale graph problems as an instance of data-intensive workloads.

On the other hand, emerging NVRAM (Non-Volatile Random Access Memory) devices, such as Flash, have positive aspects of low cost, high power efficiency, and huge capacity compared with conventional DRAM devices, as well as negative aspects of low throughput and latency. These NVRAM will greatly expand the possibility of processing extremely large-scale graphs that exceed the DRAM capacity of the nodes without significant performance degradation. However, little study has been done to answer the fundamental questions of how much size of graph data we can offload to the NVRAM while keeping the performance of graph analytics, which are considered as typical data-intensive workloads, and how to utilize NVRAM against these workloads.

To address the above issue, we introduce a graph data offloading technique using NVRAM that augment the Hybrid BFS (Breadth-First Search) algorithm [20] widely

used in the Graph500 benchmark [30], by offloading infrequent accessed graph data on NVRAM based on the detailed analysis of access patterns.

We develop NETALX, out-of-core Hybrid-BFS implementation by applying the graph data offloading technique to a highly NUMA-optimized in-core Hybrid BFS implementation called NETAL (NETwork Analysis Library) [92, 93].

We conduct performance analysis to demonstrate the utility of NVRAM for unstructured data and demonstrate extremely fast BFS execution for large-scale unstructured graphs whose size exceed the capacity of DRAM on the machine.

Experimental results of Kronecker graphs compliant to the Graph500 benchmark on a 2-way Intel Xeon E5-2690 machine with 256 GB of DRAM show that NETALX can achieve 4.14 Giga TEPS (Traversed Edges Per Second) for a SCALE31 graph problem with 2^{31} vertices and 2^{35} edges, whose size is 4 times larger than the size of graphs that the machine can accommodate only using DRAM, with only 14.99 % performance degradation. We also show that the power efficiency of NETALX achieves 11.8 Mega TEPS/W. Based on our implementation with further optimizations, we have achieved the 3rd and 4th position of the Green Graph500 list (2014 June) in the Big Data category.

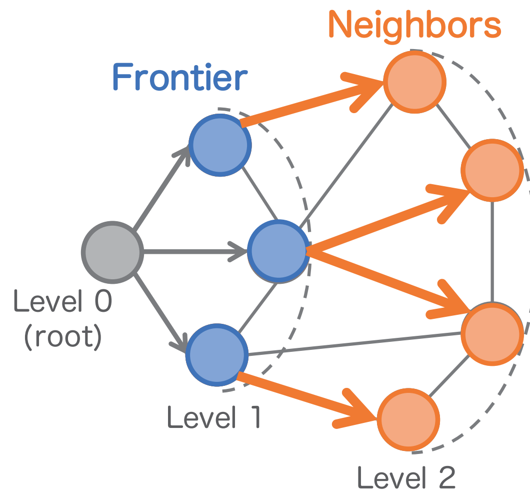
3.2 Preliminaries

3.2.1 Hybrid BFS Algorithm

The demands for high performance graph algorithms have been increasing. BFS is an important kernel of many graph algorithms. However, due to random and unstructured memory accesses and remarkably small amount of computations against I/O operations, achieving high performance on BFS is a challenging problem. Hybrid BFS algorithm [20] can drastically improve BFS performance by reducing unnecessary edge scans, so that most of modern fast BFS implementations, including that of ranked highly on the Graph500 list, employ the algorithm [84, 30]. Hybrid BFS algorithm combines two approaches, a conventional *top-down approach* and a *bottom-up approach* by changing search directions.

3.2.1.1 Top-down Approach

The top-down approach is known as a conventional BFS algorithm. Figure 3.1 shows an outline of the top-down approach with a pseudo code. Here, *frontier* denotes the set of visited vertices in a current level, *next* is the set of visited vertices at a level, *adj_vertices* denote the set of adjacent vertices of a vertex, *tree* stores the BFS tree, and *visited* holds a Boolean status if each vertex is visited. In a BFS level of the top-down approach, each vertex v in *frontier* checks all its adjacent vertices *adj_vertices(v)* to find unvisited adjacent vertices, and mark those unvisited vertices as visited in *visited* and add to the *next*.



function top-down-step(frontier, next, tree, visited)
  for $v \in$ frontier **in parallel do**
    for $w \in$ adj_vertices(v) **do**
      if visited(w) $=$ *false* **atomic then**
        visited(w) $\leftarrow$ *true*
        tree(w) $\leftarrow v$
        next $\leftarrow$ next $\cup\{w\}$
      end
    end
  end

Figure 3.1: Outline of Top-down Approach

The Drawback of Top-down Approach The top-down approach has a drawback that huge number of redundant edge checks are incurred by visiting already visited

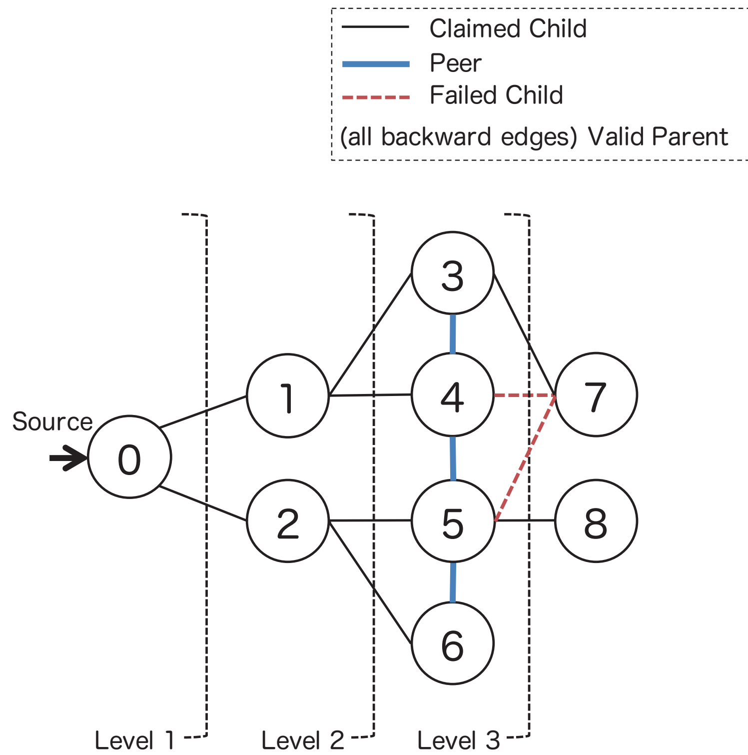


Figure 3.2: The Breakdown of Traversed Edges by Top-down Approach

vertices again. Successful cheeks for neighbor vertices result in *claimed child* category and redundant cheeks are broken into three categories: *peer*, *failed child* and *valid parent* (Figure 3.2). A *peer* is any neighbor located in the same level; a *failed child* is a vertex in next level but already visited by another vertex; a *valid parent* is a vertex in one previous level, thus, all checks for backward edges result in this category. This feature strongly appears when run BFS on small-world and scale-free graphs.

Beamer et al. reported the breakdown of edges in the frontier at each level of BFS (Figure 3.3). As you can see a large amount of redundant edge checks are performed. Note that the middle steps, 2 and 3, take most of the time of the execution.

3.2.1.2 Bottom-up Approach

The bottom-up approach traversal vertices from the reverse direction of the top-down approach. Figure 3.4 shows an outline of the bottom-up approach with a pseudo code. While the top-down approach searches for unvisited vertices from visited vertices, the bottom-up approach searches for vertices in the frontier v from all unvisited vertices

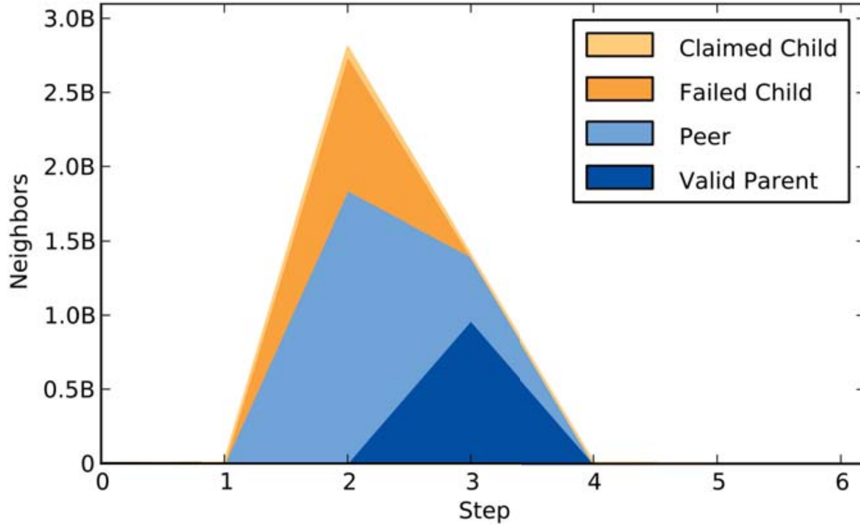


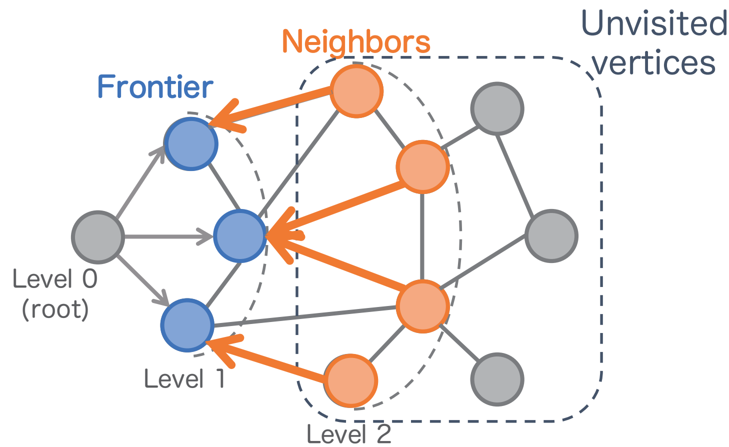
Figure 3.3: Figure from [20]; Breakdown of Edges in the Frontier for a Sample Search on kron27 (Kronecker generated 128M vertices with 2B undirected edges) on the 16-core system ("Step" corresponds to "Level")

w in a current level. If we find that a unvisited vertex w is connected to a vertex v which is included in the frontier, the vertex w is marked as "visited (w)". The bottom-up approach terminates the edge scans of each unvisited vertex w once it finds an adjacent vertex v that is in the $frontier$; thus, we can expect efficient BFS performance by this approach. However, the bottom-up approach also has a drawback that inefficient edge scans are incurred when the search state of a given graph holds only a few vertices in a frontier, since the possibility of finding vertices in frontier decreases.

3.2.1.3 Changing Search Directions

Hybrid BFS algorithm combines the benefits of the above two approaches by changing search directions. Table 3.1 shows the number of traversed edges by the Top-down (m_F), Bottom-up (m_B), and Hybrid ($oracle$) approaches. When only the top-down approach is selected in all levels, 100% of edges of the graph are traversed; selecting only the bottom-up approach results in over 180% of edge checks. However, by properly selecting a direction in each level, only 3.12% of edges are traversed and can reduce the runtime for BFS drastically.

A basic idea is that beginning BFS with the top-down approach and switch to the bottom-up approach when the size of frontier becomes too large; back to the top-down



function bottom-up-step(frontier, next, tree, visited)
    for $w \in$ vertices **in parallel do**
        if visited$(w) = false$ **then**
            for $v \in$ adj_vertices(w) **do**
                if $v \in$ frontier **then**
                    tree$(w) \leftarrow v$
                    visited$(w) \leftarrow true$
                    next $\leftarrow$ next $\cup\{w\}$
                    break
                end
            end
        end
    end

Figure 3.4: Outline of Bottom-up Approach

approach for the final steps where the size of the frontier is too small. However, the timing of switching the top-down and bottom-up approach to another approach is not obvious since computing the exact number of traversed edges by each approach takes high cost or impossible without actually performing a traversal. Thus, Beamer, S. et al. have proposed heuristic models using some parameters, e.g., the number of vertices in the frontier, the number of edges from the frontier, and the number of unvisited vertices [19, 20]. We will discuss the detail our model to select search direction in Section 3.4.4.4.

Table 3.1: Data from [92]; The number of traversed edges by Top-down (m_F), Bottom-up (m_F) and Hybrid (*oracle*) in BFS on a kronecker graph

Level	Top-down m_F	Bottom-up m_B	Hybrid (*oracle*) $\min(m_F, m_B)$
0	2	2,103,840,895	2 (m_F)
1	66,206	1,766,587,029	66,206 (m_F)
2	346,918,235	52,677,691	52,677,691 (m_B)
3	1,727,195,615	12,820,854	12,820,854 (m_B)
4	29,557,400	103,184	103,184 (m_B)
5	82,357	21,467	21,467 (m_B)
6	221	21,240	221 (m_F)
Total	2,103,820,036	3,936,072,360	65,689,625
Ratio	100.00%	187.09%	3.12%

3.3 Out-of-core Hybrid BFS

We propose NETALX, out-of-core (semi-external memory) Hybrid BFS implementation based on a highly NUMA-optimized in-core Hybrid BFS implementation called NETAL (NETwork Analysis Library) [92, 93], one of the fastest single-node Hybrid BFS implementation on the Graph500 list.

3.3.1 NUMA-Optimized Hybrid BFS Implementation

NETAL is a highly NUMA-optimized in-core Hybrid BFS implementation that carefully considers the NUMA architecture of au underlying system in order to perform highly efficient localized memory accesses to CSR graphs, *forward graph* in the top-down approach and *backward graph* in the bottom-up approach, and other data structures used in BFS.

The top-down approach explores unvisited vertices from vertices in the frontier. In general, each vertex straightforwardly holds all connected vertices; however, such naive graph representation introduces costly frequent remote memory accesses to non-local

NUMA nodes. NETAL avoids such inefficient remote memory accesses by partitioning all vertices in neighbors into a small portion of the vertices based on local NUMA nodes and by duplicating corresponding frontier vertices. For example, a source vertex in a NUMA node explores destination vertices in the same NUMA node. If destination vertices belong to different NUMA nodes, NETAL delegates the search to other source vertices that belong to the same NUMA node as the destination vertices.

On the other hand, the bottom-up approach explores vertices in the frontier from unvisited vertices. In order to provide efficient memory access to local NUMA nodes, NETAL partitions edges of vertices based on the NUMA nodes, so that candidate vertices in the frontier are stored on the same NUMA node.

3.3.2 Out-of-core Hybrid BFS with Dual Graph Model

Based on the NUMA-based optimized Hybrid BFS implementation that employs two graph data structures, the forward graph for top-down approaches and the backward graph for bottom-up approaches, to perform NUMA optimized BFS, thus a straightforward model of out-of-core Hybrid BFS is that carefully offloads infrequent accessed forward graph data to NVRAM and directly reads the forward graph data from the devices on demand, while keeping backward graph data on DRAM. Because the performance of Hybrid BFS algorithm is highly depends on the performance of bottom-up approach. Here, we call this implementation *the dual graph model*. Experimental results to a Scale 27 problem of a Kronecker graph with 2^{27} vertices and 2^{31} edges, whose size exceeds the capacity of DRAM on the node, show that our approach maximally sustains 4.22 GTEPS with only 19.18% performance degradation on a 4-way AMD Opteron 6172 machine heavily equipped with NVRAM devices.

3.3.3 Problems for Scaling Graphs Using NVRAM

Although our out-of-core Hybrid BFS with dual graph model can process large-scale graphs per node with minimum performance degradation, its implementation includes several drawbacks and problems.

3.3.3.1 Redundant Graph Data Structures

The dual graph model requires two graph data structures, called the forward graph and the backward graph, for the top-down and the bottom-up approach respectively. The model has several advantages that we can easily analyze various access patterns of different two approaches to distinguished graph data structures and straightforwardly offload infrequently accessed graph data structures onto semi-external memory devices; however, this graph model introduces redundant memory storage consumption as holding two graph data structures. Besides, the dual graph model cannot control the ratio of storage usage between DRAM and NVRAM since the size of offloading graph data, i.e., the forward graph, is fixed.

3.3.3.2 Optimized Data Allocation and I/O Strategies

We may merge the distinguished two graph data structures to a single graph data structure, whereas we have to organize appropriate graph data allocations and accesses onto hierarchal memory in order to minimize performance degradation. Indeed, performance degradation is mainly caused by large amounts of fine-grained small I/O operations to semi-external memory devices.

3.4 Out-of-core Hybrid BFS with Memory Efficient Data Structure

3.4.1 Hybrid BFS with Single Graph Model

As described in Section 3.3.2 and 3.3.3, in order to avoid the drawback of redundant graph data structures of our dual graph model implementation, we introduce a memory efficient Hybrid BFS implementation that merges the two graph data structures to a single graph data structure using *a backward graph* based on the dual graph model [92];

Here, we call this implementation *the single graph model*. Main differences between these two models, the dual graph model and the single graph model, are derived from the implementation in *the top-down approach*. In the dual graph model, destination vertices are accessed from source vertices are stored on the same NUMA node as the destination vertices. In order to achieve efficient search, frontier is duplicated across the NUMA nodes based on the configuration of the graph. On the other hand, in the single graph model, frontier is divided into the NUMA nodes of the underlying machine, and each vertex in frontier accesses to destination vertices that may belong to different NUMA nodes. Thus, the single graph model of Hybrid BFS implicitly includes performance degradation derived from remote NUMA node accesses. However, because the performance of hybrid BFS is delivered from the bottom-up approach, we believe actual performance degradation of this model is small.

3.4.2 Analysis of Data Accesses in Bottom-up Approach

The bottom-up approach terminates edge scans once an edge connected to the frontier is found. In order to elucidate the access patterns of the bottom-up approach, we firstly investigate the number of edges each vertex scans, on a SCALE31 graph problem. Figure 3.5a shows that how many edges a source vertex scans on average. The x-axis denotes the number of edges m that a source vertex scans, and the y-axis with a logarithmic scale denotes the number of source vertices that scan m edges. This result indicates that most of source vertices scan only a few edges. For example, 48.9 billions of source vertices scan a single edge, while only 70 source vertices scan 60 edges; moreover, no source vertices scan over 67 edges. Thus, if we accommodate at least 67 edges for each source vertex on DRAM, we can achieve highly efficient BFS performance.

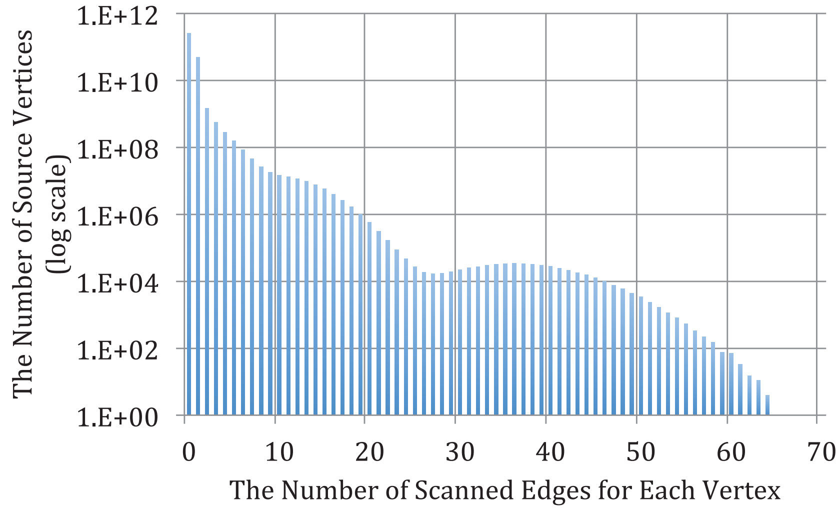
BFS performance is actually affected by the number of *traversed edges*. Thus, we analyze the number of traversed edges based on the number of edges that a source vertex scan in all bottom-up approach performed in a single BFS. Figure 3.5b shows the result, where the x-axis denotes the number of scanned edges of a source vertex, the *left* y-axis denotes the number of traversed edges by source vertices, and the *right* y-axis denotes the integral of the completion ratio of traversed edges obtained by dividing the number of traversed edges by source vertices by the total number of traversed edges. Note that the values of the *left* y-axis in Figure 3.5b are obtained from the product

of values of the x-axis and the y-axis in Figure 3.5a. The result indicates that we can complete most of edge scans in a given graph to finish BFS if each source vertex scans only a few edges; 84% of edge traversals are completed when each of source vertices scan a single edge, and 90% of edge traversals when 2 edges. Finally, 99.95% of edge traversals are completed when each of source vertices scan at maximum 21 edges, although the average degree of a given graph is 16 and the maximum degree of that is $9,305,421$.

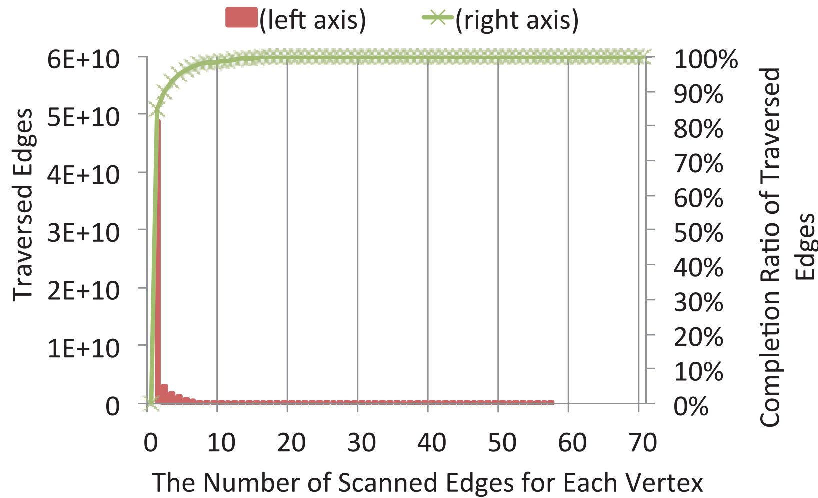
Based on the above results, we can expect to reduce memory consumption by offloading a part of graph data onto semi-external secondary memory without significant performance degradation.

3.4.3 Out-of-core Hybrid BFS with Single Graph Model

The intuitive idea of the single graph model of the out-of-core Hybrid BFS algorithm is that we offload infrequent accessed data sets onto secondary devices and read the data sets directly from the secondary devices on demand. To do so, we hold a given number of edges per vertex on DRAM and offload remaining edges onto NVRAM as files. In the top-down approach, all edges stored on both DRAM and NVRAM have to be scanned to search unvisited vertices. On the other hand, in the bottom-up approach, we firstly scan edges on DRAM to find a vertex in the frontier. If we find such a vertex, we terminate the edge scan and move on to the next step. Otherwise, we continue to the scan edges on NVRAM to find a vertex in the frontier. In order to increase the probability for finding the frontier vertices in the bottom-up approach, we sort the edges of each vertex in descending order of the degree of destination vertices, as the same technique as [93], and preferentially store high degree vertices on DRAM. Thus, we can expect to achieve highly efficient edge scans by reducing fine-grained small I/O operations to NVRAM, while saving redundant DRAM consumption. In addition, the single graph model of out-of-core Hybrid BFS has another advantage that we can flexibly control the size of graph data to store on NVRAM by changing the maximum number of edges per vertex to store on DRAM.



(a) Number of Source Vertex that Scans m Edges



(b) Number of Traversed Edges by Source Vertex and Completion Ratio

Figure 3.5: Analysis of Data Accesses in Bottom-up Approach

3.4.4 Implementation Details

3.4.4.1 Data Structures

The single graph model of out-of-core Hybrid BFS consists of a backward graph represented by the CSR (Compressed Sparse Row) format and BFS status management data, i.e., bitmaps and tree for keeping BFS status. Figure 3.6 shows an overview of the

data structures of the single graph model. A graph in the CSR graph format consists

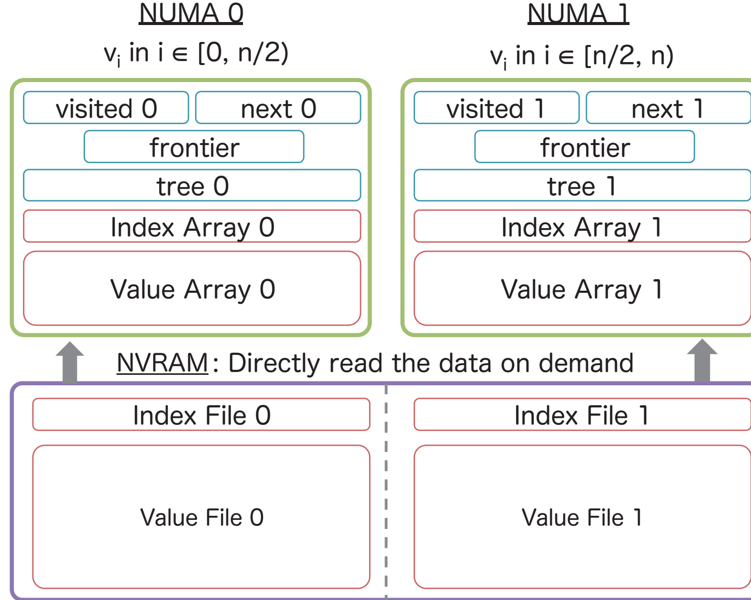


Figure 3.6: Data Structures of Out-of-core Hybrid BFS with Single Graph Model

of two array structures called the *index* array and the *value* array. The *index* array stores indices of the *value* array, and the *value* array stores vertex IDs; each index in the *index* array represents a source vertex, and the corresponding element in the *index* array refers to an index in the *value* array. The *value* array stores IDs for destination vertices. In our implementation, we keep a part of the *index* and *value* arrays on DRAM and store the entire *index* and *value* arrays on NVRAM as corresponding files, which we call the *index* file and the *value* file. In the top-down approach, we read 4KB of a continuous region for a vertex by using POSIX read(2) API, since we have to access all adjacent vertices, whereas, in the bottom-up approach, we read 512B of small chunks by using POSIX read(2) API in order to avoid unnecessary small I/O operations. BFS status management data consist of 3 bitmaps and 1 array: a bitmap to store vertices in the frontier in the current level (*frontier*), a bitmap to store next level frontier (*next*), and a bitmap to manage visited vertices (*visited*); and an array to memory a BFS tree by storing parent vertex IDs (*tree*).

The total required data size for a graph is calculated as follows. Let n be the number of vertices, and m be the number of edges. Vertex IDs are represented by using 64-bit integer. Therefore, the size of the index array is calculated as $8 * (n + 1)$ bytes, including terminator, and the size of the value index is calculated as $8 * 2 * m$ bytes. Note that here we focus on an undirected graph, so that we have to store both incoming and outgoing edges for a vertex. *tree* stores vertex IDs with 64-bit integer,

so the size of *tree* is calculated as $8 * n$ bytes. The size of *next* and *visited* is calculated as $n/8$ bytes, since these data are represented as bitmaps. *frontier* is represented as a bitmap and duplicated across the NUMA nodes in the system, so the size of *frontier* is calculated as $n/8 * l$ bytes, where l denotes the number of NUMA nodes.

3.4.4.2 Graph Data Partitioning Based on NUMA Architecture

In the backward graph, vertices are divided by considering NUMA nodes of the underlying system. Let n be the number of vertices, v_i be a vertex with $i \in \{0, \cdots, n-1\}$ as a vertex ID, ℓ be the number of NUMA nodes in the system, and N_k be the kth NUMA node with $k \in \{0, \cdots, \ell - 1\}$. In the above setting, we group vertices v_i in $i \in \left[k \cdot \frac{n}{\ell},\ (k+1) \cdot \frac{n}{\ell}\right)$. Then, a part of edges of vertices v_i are kept on a NUMA node N_k and the entire edges are stored on the files on NVRAM based on the NUMA node k. BFS status management data, i.e., *next*, *visited* and *tree*, are also basically divided across the NUMA nodes as the same manner, whereas *frontier* is duplicated across the NUMA nodes in order to provide highly efficient localized memory accesses. Specifically, in the bottom-up approach, when a thread is assigned to a NUMA node, all data sets, including a backward graph and BFS status management data, can be localized in the same NUMA node, so that threads assigned on the NUMA node can achieve extremely fast read and write operations. On the other hand, in the top-down approach, write operations may also be conducted to BFS status management data, such as *next*, *visited*, and *tree*, on remote NUMA nodes, since these BFS status management data are distributed across NUMA nodes. Therefore, performance degradation may be occurred, although read operations from a backward graph and BFS status management data such as *frontier* are localized in the same NUMA node.

3.4.4.3 Edge Scan

Here, we describe outlines of the single graph model of the out-of-core Hybrid BFS algorithm for the top-down approach in Algorithm 1 and for the bottom-up approach in Algorithm 2, where *Idx-DRAM* denotes the index array on DRAM, *Val-DRAM* denotes the value array on DRAM, *Idx-NVRAM* denotes the index file on NVRAM,

and *Val-NVRAM* denotes the value file on NVRAM.

Given a current level in the top-down approach, we first select a vertex v in the frontier and read an adjacent vertex w from the *Val-DRAM* value array. If w is marked as "unvisited (0)", we mark the vertex w as "visited (1)" and then the vertex w are added to the *next*. Similarly, we read adjacent vertices on NVRAM from the *Val-NVRAM* value file contiguously in 4KB each and conduct the marking process. After checking all vertices in the *Val-DRAM* value array and the *Val-NVRAM* file, we move on to the next search process by replacing vertices in the frontier.

On the other hand, in the bottom-up approach, we search for vertices in the frontier from all unvisited vertices w in a given current level. To do so, we read a vertex v adjacent to an unvisited vertex w from the *Val-DRAM* value. Then, if a vertex v on DRAM is not included in the frontier, we read remaining vertices from NVRAM in 512B of contiguous chunks. Note that we can skip to search vertices v once we find that an unvisited vertex w is connected to a vertex v in the frontier. After finding the vertex v in the frontier, we mark the unvisited vertex w as "visited (1)".

3.4.4.4 Changing Search Directions

Our proposed implementation changes search directions based on the assessment of the expected number of vertices and edges. Let parameters α and β be the thresholds for changing search directions, i be a given level, $n_{f(i)}$ be the number of vertices in the frontier, $n_{u(i)}$ be the number of unvisited vertices, $m_{td(i)}$ be the expected number of scanned edges using the top-down approach, and $m_{bu(i)}$ be the expected number of scanned edges using the bottom-up approach. Here, we obtain $m_{td(i)}$ as $m_{td(i)} = edge_factor \times n_{f(i)}$ and $m_{bu(i)}$ as $m_{bu(i)} = edge_factor \times n_{u(i)} + n_{f(i)}$. Based on these parameters, we use the bottom-up approach when $n_{f(i-1)} < n_{f(i)}$ and $m_{td(i)} > m_{bu(i)}/\alpha$, while we use the top-down approach when $n_{f(i-1)} > n_{f(i)}$ and $m_{td(i)} < m_{bu(i)}/\beta$. Note that α is used to change to the bottom-up approach, while β is used to return to the bottom-up approach. We set $\alpha = 4096$ and $\beta = 4$ in the implementation determined by the preliminary experiments.

```
for v ∈ frontier in parallel do
    for i ∈ range (Idx-DRAM(v), Idx-DRAM(v+1)) do
        w ← Val-DRAM(i)
        /* if w is not visited */
        if visited(w) = 0 atomic then
            tree(w) ← v
            visited(w) ← 1
            next ← next ∪{w}
        end
    end
    for i ∈ range (Idx-NVRAM(v), Idx-NVRAM(v+1)) do
        w ← Val-NVRAM(i)
        /* if w is not visited */
        if visited(w) = 0 atomic then
            tree(w) ← v
            visited(w) ← 1
            next ← next ∪{w}
        end
    end
end
```

Algorithm 1: Outline of NVRAM-Based Top-down Approach

3.4.4.5 Out-of-core Graph Construction

In this section, we describe how to construct a large-scale graph of which size exceeds DRAM capacity. A naive approach to construct a graph converting a list of edges into a CSR is as follows: 1) sequentially read edges and count the degrees of all vertices; 2) place each edge into an appropriate position in the edge array of the CSR graph while referring and updating the index array constructed in the previous step. This approach works well on in-core situation; however, significant performance degradation will happen on out-of-core processing situation because constructing a graph with CSR format with the approach causes a lot of random and fine-grained I/O operations.

Therefore, we construct a graph using a different method. An overview of our

```
for w ∈ vertices \ visited in parallel do
    for i ∈ range (Idx-DRAM(w), Idx-DRAM(w+1)) do
        v ← Val-DRAM(i)
        if v ∈ frontier then
            tree(w) ← v
            visited(w) ← 1
            next ← next ∪{w}
            goto Next-Vertex
        end
    end
    for i ∈ range (Idx-NVRAM(w), Idx-NVRAM(w+1)) do
        v ← Val-NVRAM(i)
        if v ∈ frontier then
            tree(w) ← v
            visited(w) ← 1
            next ← next ∪{w}
            goto Next-Vertex
        end
    end
    Next-Vertex:
end
```

Algorithm 2: Outline of NVRAM-Based Bottom-up Approach

out-of-core graph construction method is illustrated in Figure 3.7. Its intuitive idea is that 1) divide the edgelist into multiple chunks so that the size of each subgraph doesn't exceed the size of DRAM after constructing it; 2) for each chunk of the edgelist, construct a CSR graph and dump it into external memory; 3) finally, merge the CSR graphs to construct the full size of CSR graph. When perform the approach above, I/O operations between DRAM and external memory occurs multiple times; however, those I/O operations will be sequential memory accesses. Thus, performance degradation result in low.

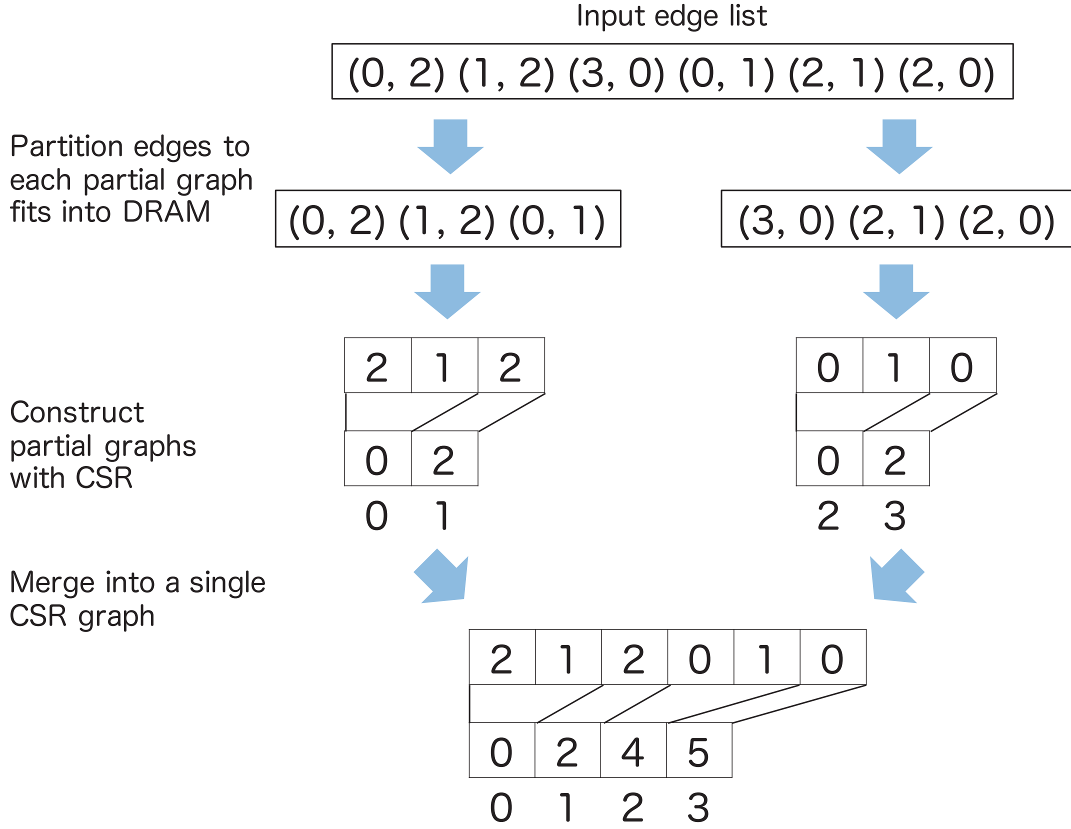


Figure 3.7: Out-of-core Graph Construction

3.5 Experiments

In order to evaluate our memory efficient implementation for the out-of-core Hybrid BFS algorithm and demonstrate the utility of NVRAM for unstructured graph data, we conduct experiments of out-of-core BFS.

3.5.1 Experimental Setup

We set the following two scenarios to evaluate the performance of our graph offloading technique. **DRAM-only**: all data is stored on DRAM (NETAL). **DRAM+NVRAM**: data is stored on DRAM and NVRAM by using our proposed technique (NETALX). In this experiment, we use Kronecker graphs with 2^{SCALE} edges with edge_factor 16 in the Graph500 benchmark. The details of the machine

CPU	Intel(R) Xeon(R) CPU E5-2690 @ 2.90GHz (8 cores, 16 threads) × 2 sockets
DRAM	256GB
NVM	EBD-I/O 2.0TB × 2 File system : EXT4, I/O scheduler : noop
OS	CentOS 6.4 (kernel 3.12.6)

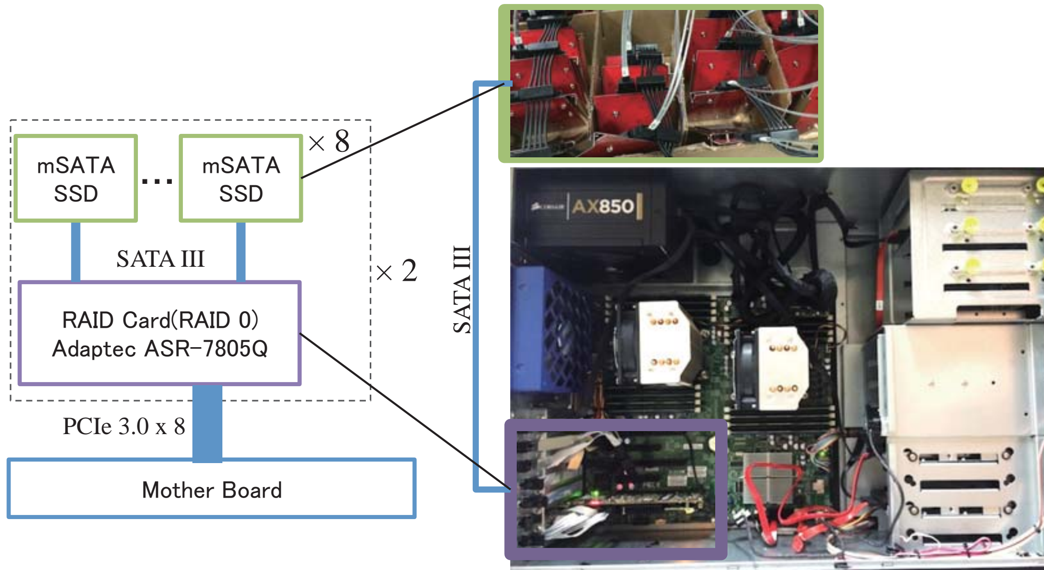


Figure 3.8: Overview of EBD-I/O device

configuration are shown in Figure 3.8. Combining multiple NVRAM devices contirbutes to improve bandwidth and IOPS performance without using expensive NVRAM devices [96].

Here, we use a special self-customized NVRAM component, called the EBD (Extreme Big Data)-IO device, for semi-external memory devices, which aggregates multiple mini SATA (mSATA) SSDs by using a RAID card in order to achieve high throughput and high IOPS performance as well as huge storage capacity at a low cost and in a small footprint [80, 76]. Specifically, we use 2 units of EBD I/O devices each of which consists of 8 devices of Crucial mSATA SSD 256 GB and a single Adaptec ASR-7805Q RAID controller card. We configure the volume as RAID0 with the EXT4 file system with the noop I/O scheduler.

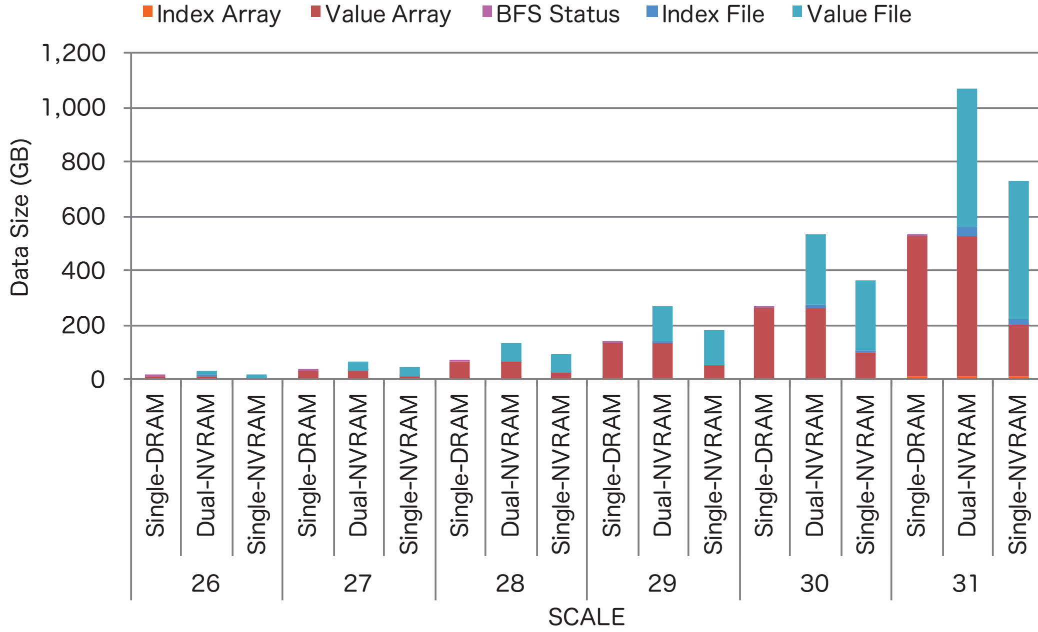


Figure 3.9: Comparison of Memory Consumption

3.5.2 Memory Consumption

First of all, we compare the memory consumption of our three implementations in Figure 3.9: **Single-DRAM**: NUMA-optimized in-core Hybrid BFS [92] with Single Graph model; **Dual-NVRAM**: out-of-core Hybrid BFS with Dual Graph Model; and **Single-NVRAM**: out-of-core Hybrid BFS with Single Graph Model (proposal). Note that the index and value files are stored on NVRAM. Here, we assume that Single-NVRAM stores 37% of the entire graph on DRAM. We see that our proposed technique drastically reduces DRAM memory consumption; Single-NVRAM achieves 2.6 times DRAM memory efficient than Dual-NVRAM for a SCALE31 graph. Note that, since the experimental machine has 256 GB of DRAM memory, the maximum problem size that the machine can accommodate is SCALE29.

3.5.3 BFS Performance for Large-scale Graphs

In order to investigate BFS performance for large-scale graphs, we vary the SCALE parameter of Kronecker graphs from 26 to 31, while keeping the edge_factor parameter

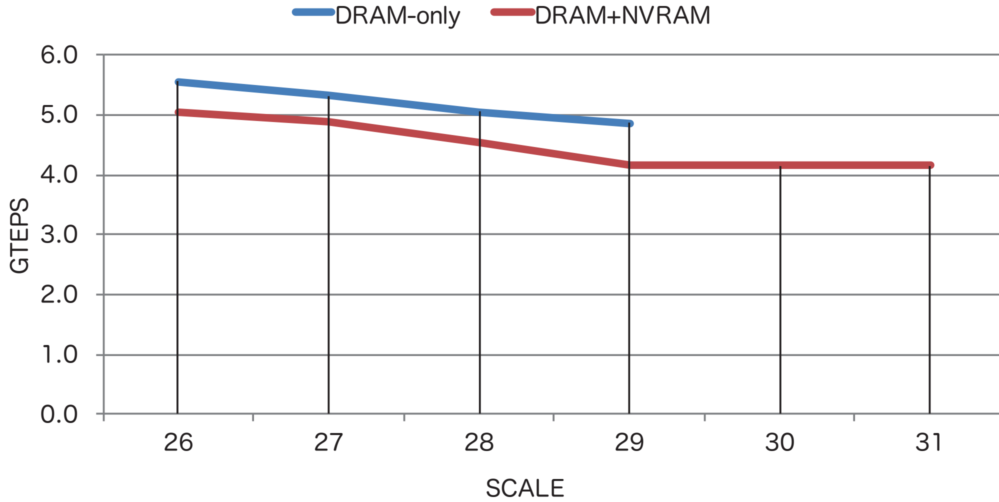


Figure 3.10: BFS Performance in TEPS

to 16. Figure 3.10 shows the performance results in TEPS. We see that DRAM-only achieves 4.87 GTEPS for a graph with the maximum problem size (SCALE29) that fits the capacity of DRAM on the machine. On the other hand, DRAM+NVRAM can process large graphs whose size exceed the capacity of DRAM on the machine; we achieve 4.14 GTEPS for the SCALE31 problem graph, whose size is 4 times larger than the maximum graph size of the DRAM-only configuration, with only 14.99 % performance degradation.

To clarify the performance degradation, we also investigate the access patterns of the results in Figure 3.10. In the top-down approach in DRAM+NVRAM, 90% of memory accesses for SCALE30 and 99% of memory accesses for SCALE31 are conducted to NVRAM. This is because the top-down approach has to scan all edges a source vertex has, while only a part of edges is stored on DRAM and the remains are stored on NVRAM. On the other hand, in the bottom-up approach in DRAM+NVRAM, only a few accesses are conducted to NVRAM, since the bottom-up approach does not need entire edge scans due to the algorithm properties.

Next, we also analyze the reason why the performance overheads in the top-down approach that conducts massive amounts of accesses to NVRAM is small. Since all edges connected from a source vertex have to be scanned, we read maximally 4KB of a chunk at once. Thus, our approach can conduct efficient I/O operations over 512 edges, assuming 8 bytes for a single edge. In order to investigate how large we can read continuous regions from NVRAM, we monitored the average number of edges read

from NVRAM for each source vertex during the top-down approach. The results show that our approach conducts I/O operations to large continuous regions on NVRAM, reading 52317.4 edges for SCALE30 and 29710.9 edges for SCALE31 on average from NVRAM. Therefore, we can reduce access latency to slow NVRAM devices and achieve fast BFS performance, although we read massive amounts of edges from NVRAM in the top-down approach.

3.5.4 BFS Performance on Reduced Memory Configuration

In order to evaluate how much we can reduce memory consumption for performing BFS using our proposed technique, we measure BFS performance when we change the number of partial edges to store on DRAM under the fixed graph size. Figure 3.11 shows the results for a SCALE31 graph problem, where the x-axis denotes the maximum number of partial edges stored on DRAM for each vertex, the *left* y-axis denotes the BFS performance in TEPS, and the *right* y-axis denotes the size of graph data on DRAM. The results show that BFS performance basically increases according to the number of edges stored on DRAM; we can maximally sustain 4.14 GTEPS for storing 256 edges per vertex and using 206.84 GB of required data in total on DRAM. However, the performance is saturated when we increase the number of edges to store on DRAM. For example, we can also achieve 4.07 GTEPS for using 64 edges per vertex and 137.54 GB of required data on DRAM, whose result shows only 1.7% of performance differences compared with the best case.

In order to clarify the cause of the performance we achieved in Figure 3.11, we investigate the breakdown of the execution times in the both top-down and bottom-up approaches for a SCALE31 graph. Figure 3.12 shows the results, where the x-axis denotes the maximum number of partial edges stored on DRAM for each vertex, and the y-axis denotes the average execution times in the both approaches of 64 times BFS trials. The results show that the execution times for the top-down approach exhibit constant behavior, since the time for accesses to NVRAM devices dominates the total execution time for the top-down approach; 29710.9 edges are read from NVRAM for each vertex on average. Thus, the time for accesses to small numbers of edges on DRAM affects negligible performance improvement. On the other hand, the execution times for the bottom-up approach drastically decrease, so we investigate the number

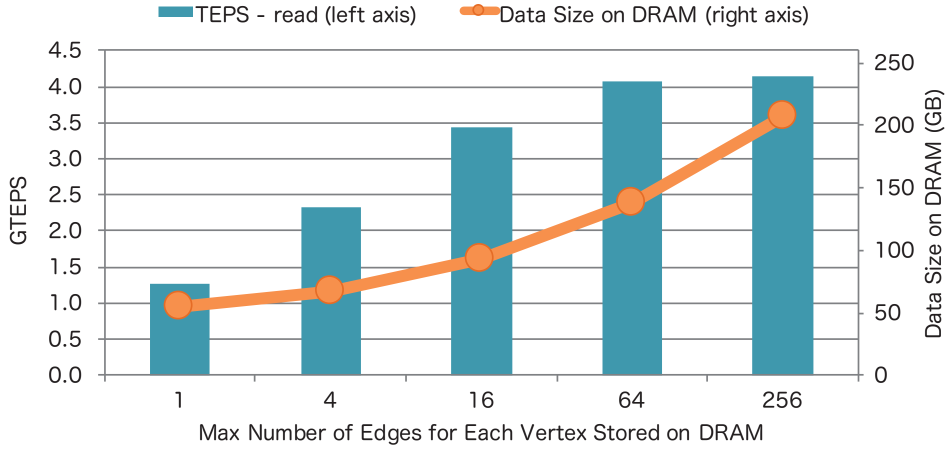


Figure 3.11: BFS Performance and Memory Consumption

of accesses to the index and value files on NVRAM during the bottom-up approach. Figure 3.13 shows the results for a SCALE31 graph, where the x-axis denotes the maximum number of edges for each vertex on DRAM, and the y-axis denotes the number of accesses in NVRAM. Here, when we read n edges for a source vertex, we count 1 access to the index file to get a source vertex and n accesses to the value file to get destination vertices. When we increase the number of edges on DRAM, we observe that the accesses to the index and value files drastically decrease; we can almost omit accesses to the index and value files when we hold over 64 edges for each vertex on DRAM; we see negligible performance differences between the configurations of storing 64 edges and 256 edges on DRAM. Therefore, we can achieve extremely fast BFS by using NVRAM devices.

3.5.5 BFS Performance Changing Edge_factor Parameters

In order to evaluate robustness of our proposed technique, we conduct BFS execution to Kronecker graphs with various edge_factor parameters. Figure 3.14a and Figure 3.14b show the results in TEPS to given graphs that the machine can maximally accommodate by changing the edge_factor parameter from 16 to 64. Note that, in the DRAM-only configuration, the maximum graph sizes that the machine can accommodate are SCALE29 with edge_factor 16, SCALE28 with edge_factor 32, and SCALE27 with edge_factor 64, while, in the DRAM+NVRAM configuration, the

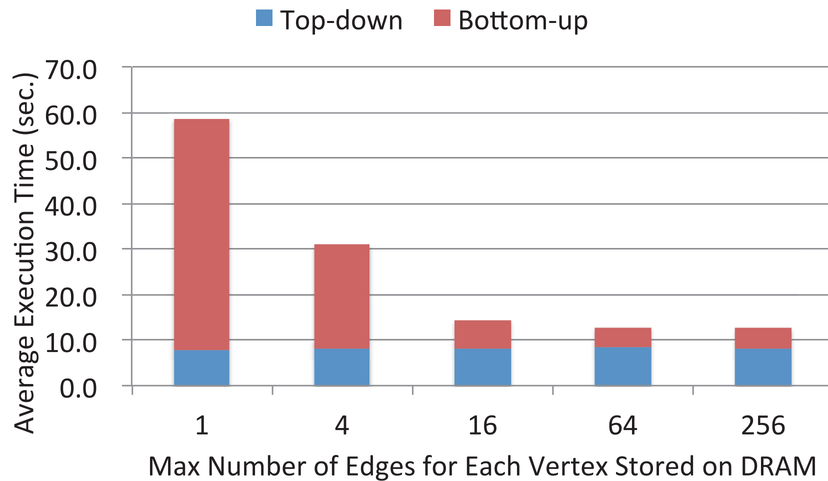
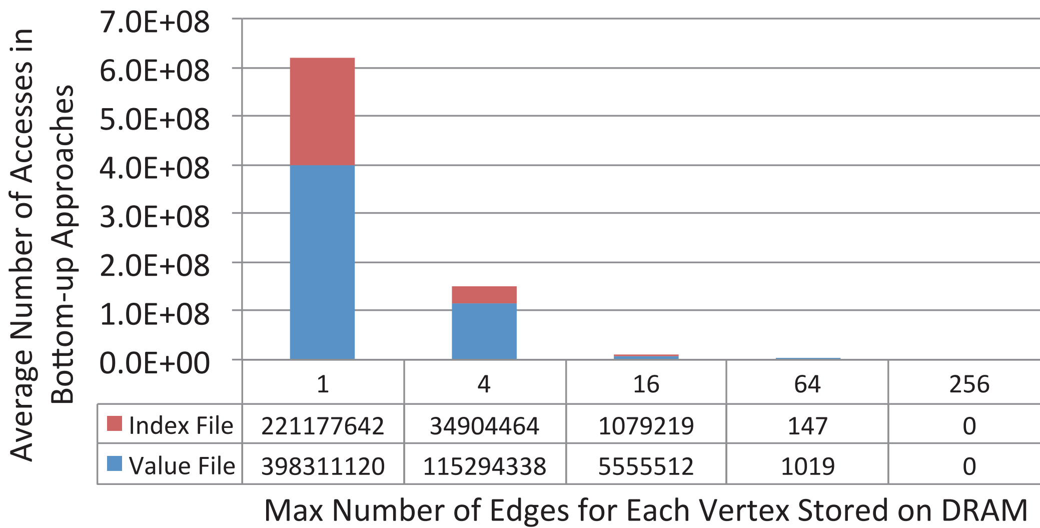


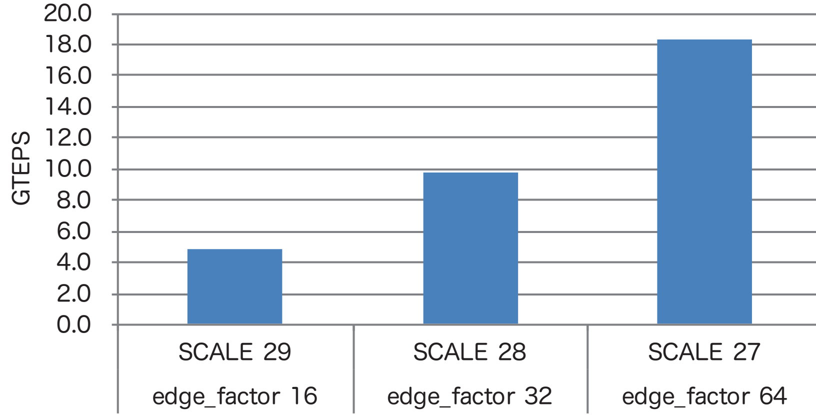
Figure 3.12: Average Execution Times in Both Approaches



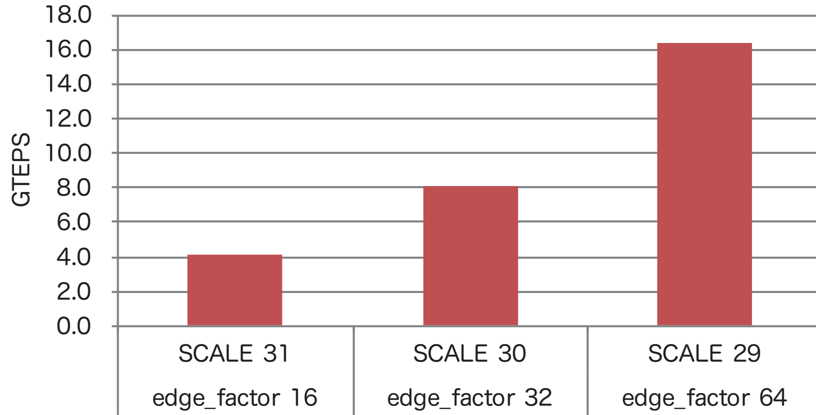
	1	4	16	64	256
Index File	221177642	34904464	1079219	147	0
Value File	398311120	115294338	5555512	1019	0

Figure 3.13: Number of Accesses to NVRAM in Bottom-up Approach

maximum graph sizes are SCALE31 with edge_factor 16, SCALE30 with edge_factor 32, and SCALE29 with edge_factor 64. The results show that, as the edge_factor parameter increases, we obtain better BFS performance in both configurations; DRAM+NVRAM achieves approximately 90% of BFS performance for a 4 times large graph with edge_factor 64 compared to DRAM-only. In the Hybrid-BFS algorithm, when the number of edges in a given graph increases, especially with scale-free properties and small diameters, we can drastically improve BFS performance, since the bottom-up approach significantly cut a lot of redundant edge scans. We see that our proposed technique (DRAM+NVRAM) also shows similar



(a) DRAM-only



(b) DRAM+NVRAM

Figure 3.14: BFS Performance in TEPS with changing edge_factor

performance properties to the original Hybrid-BFS algorithm in the DRAM-only configuration, even when we change the edge_factor parameter in a given graph, and their size exceeds the capacity of DRAM on the machine.

3.5.6 Power Efficiency Using NVRAM

Our proposed technique can also contribute power efficient BFS execution for large-scale graphs. In order to verify the claim, we conduct the Green Graph500 benchmark, which measures average power consumption during BFS execution in Graph500 and scores performance-per-watt metrics in TEPS per Watt (TEPS/W), on the both DRAM-only and DRAM+NVRAM configurations. Figure 3.15 shows

the results of the average power consumption (W) and Figure 3.16 shows the power efficiency metric (MTEPS/W) for SCALE29, 30, and 31 graphs. We see that DRAM-only exhibits higher power consumption and higher power efficiency than DRAM+NVRAM: 367.4 W and 13.2 MTEPS/W for DRAM-only and 340.5 W and 12.2 MTEPS/W for DRAM+NVRAM for a Scale 29 graph. In DRAM-only, all graph data accesses are conducted on DRAM, so that we can achieve higher BFS performance than DRAM+NVRAM, whereas the power consumption also increases. On the other hand, DRAM+NVRAM achieves similar power consumption and energy-efficiency metric results: 11.8 MTEPS/W, although our proposed technique can solve 4 times larger graphs whose size exceed the capacity of DRAM on the machine.

Based on our proposed technique and further optimizations, we have achieved the 3rd and 4th position on the Graph Graph500 (June 2014) in the Big Data category using a single server. Specifically, we used a newer version of NETAL that possesses 3x better BFS performance because of further optimizations such as removing 0 degree vertices before constructing a graph; sorting the order of vertices depend on these degree; storing the highest degree adjacent edge for each vertex into a contiguous memory region. In addition, we also applied a few optimizations for NETALX: allocating the index file into DRAM instead of NVRAM; changing its file I/O interface to mmap system call.

3.6 Related Work

GraphChi [53] uses a latency hiding technique that divides a large graph into small chunks and sequentially reads them using Parallel Sliding Windows (PSW) method, in order to compute large-scale graphs on a single node; X-Stream [74] employs an edge-centric rather than a vertex-centric computation model. However, because all edges are streamed in every iteration, these techniques lead to a large number of unnecessary reads for BFS.

Pearce et al. have proposed NVRAM-based graph kernels, including BFS, for processing extremely large-scale graphs by hiding access latencies with the help of massive amounts of asynchronous multithreads [69]. To utilize the SSDs' properties, FlashGraph [97] only accesses required data and conservatively merges I/O requests to increase I/O throughput and reduce CPU overhead for I/O. G-store [51] employs

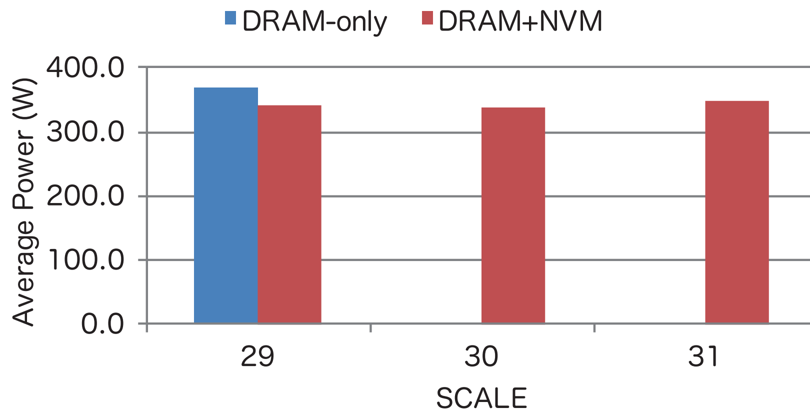


Figure 3.15: Average Power Consumption

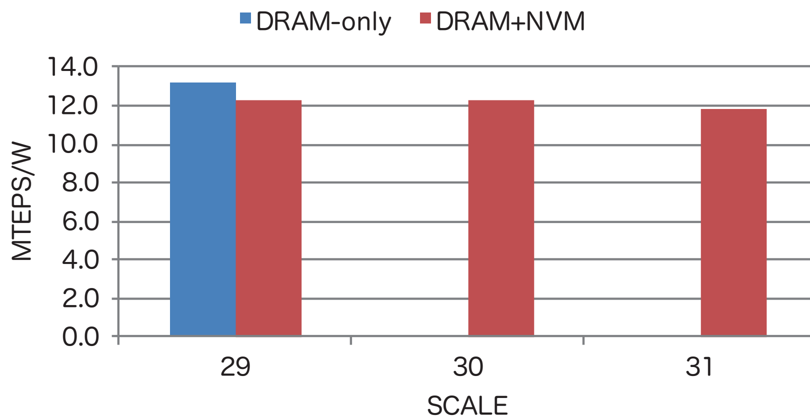


Figure 3.16: Power Efficiency Metric in MTEPS/W

small number of bits to represent vertex ID to reduce total graph size and locality-aware graph processing models. Although NETALX differs in that it expressly arranges partial data into DRAM to achieve extremely high performance on BFS, these techniques can be adapted to NETALX to accelerate its performance furthermore.

As for power efficiency without using NVRAM, Imamura et al., proposed a power efficient DRAM row buffer locality-aware address mapping technique for graph data structure for Hybrid-BFS algorithm based on memory access patterns in bottom-up [77].

Brian et al. have proposed a mmap implementation for NAND-flash based NVRAM and accelerated performance on graph processing [86].

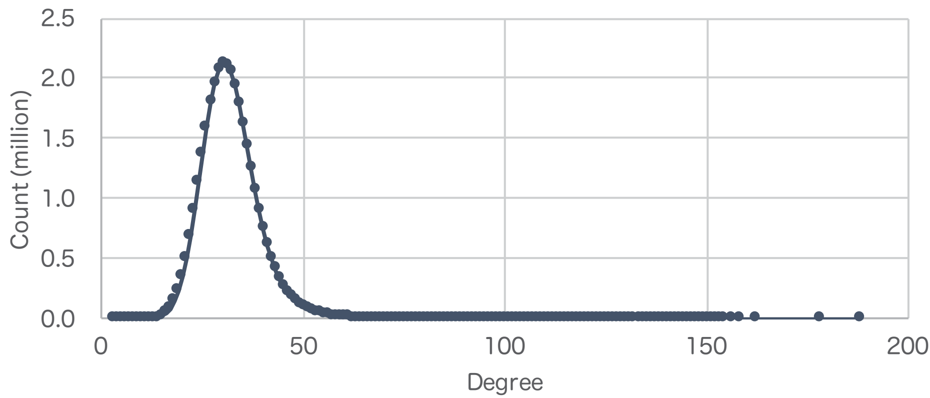


Figure 3.17: The Degree Distribution of a RMAT-ER graph with $edge_factor = 16$

As for distributed implementation of Hybrid BFS, related work is can be found in [22, 30, 50]. Pearce et al. [70, 71] have also proposed a scaling technique of graph kernels with NVRAM.

3.7 Discussion: Performance Evaluation on Normal (not scale-free) Graph

Although most of recent research regarding large-scale graph analytics are targeting scale-free graphs, and our work is also following the same path, it is worth evaluating the performance of NETALX on graphs which doesnt have a scale-free property.

To perform the evaluation, we generated a random graph using a RMAT graph generator setting its probabilities to $(0.25, 0.25, 0.25, 0.25)$ and its $edge_factor$ to 16. The generated graphs belong to the class of (*Erdős-Rényi graph*) random graphs and exhibits normal degree distribution. The degree distribution of a generated graph (RMAT-ER) is shown in Figure 3.17. As can be seen, most of vertices have around 30 edges and the maximum degree is less than 200. This degree distribution is completely difference from that of scale-free graphs.

The BFS performance on the graph with the same experimental setup described in before are shown in Figure 3.18. As the total number of edges of the graph has is same even change the distribution of degree, DRAM-only can't process graphs larger than SCALE 29 and achieves 1.1 GTEPS on a SCALE 29 graph. NETALX allocates

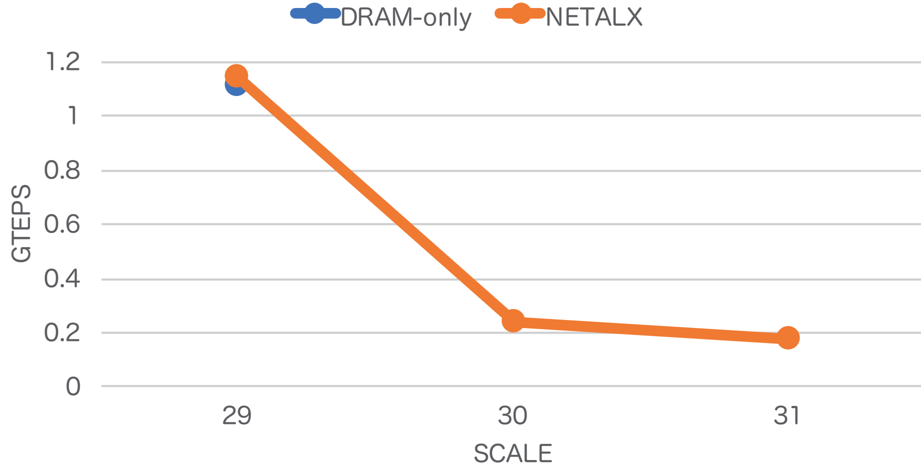


Figure 3.18: Out-of-core BFS Performance on a RMAT-ER Graph

up to 8 edges for each vertex into DRAM and achieves 0.18 GTEPS on a SCALE 31 graph (6.18x slows down against the result of DRAM-only with the SCALE 29 graph). Because the degrees of vertices whose all edges are not allocated in DRAM is not so large that NETALX can perform efficient I/O when reading adjacent edges of a vertex, its BFS performances resulted in such large slow down compared with the results we achieved in the experiment on the scale-free graphs.

Although scale-free property is found in many large-scale real graphs, and there is plenty of demands for graph analytics on such graphs, we would like to address exploring furthermore out-of-core technique for normal graphs as one of our future work.

3.8 Summary

We proposed NETALX, an extremely high performance BFS implementation using NVRAM for Hybrid BFS algorithm which deploys frequently-accessed graph data with fine-grained I/O size into DRAM space based on 1) the detailed analysis of memory access patterns of the algorithm; 2) properties of NAND flash-based NVRAM devices, i.e., fine-grained I/O causes huge overhead.

Experimental results of Kronecker graphs compliant to the Graph500 benchmark on a 2-way Intel Xeon E5-2690 machine with 256 GB of DRAM showed that our

proposed implementation with efficient memory data structures was able to achieve 4.14 Giga TEPS (Traversed Edges Per Second) for a SCALE31 graph problem with 2^{31} vertices and 2^{35} edges, whose size is 4 times larger than the size of graphs that the machine can accommodate only using DRAM, with only 14.99 % performance degradation.

We also showed that the power efficiency of NETALX achieves 11.8 Mega TEPS/W. Based on the implementation with further optimizations, we have achieved the 3rd and 4th position of the Green Graph500 list (2014 June) in the Big Data category.

Although experiments are only conducted to Kronecker graphs, the effectiveness of the Hybrid BFS algorithm to other graphs with scale-free properties and small diameters is verified in [20]. We expect that our proposed technique also has the similar algorithmic properties. Our future work includes further performance studies on various NVRAM devices and scalable implementations for multi-node environments based on our proposed techniques.

Chapter 4

Large-Scale Dynamic Graph Data Store

In many real-world graph applications, the structure of the graph changes dynamically over time and may require real-time analysis. In this chapter, we present a novel high performance dynamic graph data store (*DegAwareRHH*), which adopts an open-addressing linear-probing compact hash table that exhibit high data locality properties for especially vertices with large degree, in order to minimize the number of random accesses, reducing cache and page misses. DegAwareRHH is *degree aware*, and uses separated compact data structures for low-degree vertices to reduce their storage and search overheads.

We implemented DegAwareRHH for distributed-memory using an *asynchronous visitor queue abstraction* [70] [71], and perform scaling studies of large scale graph construction on up to 192 compute nodes.

Summary of our contributions of this chapter:

- We demonstrate that DegAwareRHH can process 1 billion edge insertion requests and 5% of additional edge deletion requests 212.2 times faster than STINGER[39], a state-of-the-art shared-memory streaming graph processing framework, when both implementations use 24 threads/processes.
- We present scaling studies of constructing large-scale real-world graphs including a massive-scale hyperlink graph which has 128 billion edge insertion requests, and show DegAwareRHH processes over 1.8 billion edge insertion requests per second

at 192 compute nodes.

- We also show DegAwareRHH can accelerate the performance of a massive-scale dynamic graph colouring algorithm.
- Finally, we demonstrate that DegAwareRHH also achieves high performance on out-of-core workloads including future NVRAM devices by using a NVRAM emulator.

4.1 Preliminary

4.1.1 Dynamic Graph Analytics

The design of our graph data store is targeting *incremental (streaming)* dynamic graph analytics workload; that is, actual graph analytics algorithms will be triggered when graph update requests come to the system. On the other hand, another approach, called *snapshot (batch)* model, perform graph analytics algorithms to a *snapshot* of the graph. While *incremental* dynamic graph analytics models can monitor the changes of the graph step by step, *snapshot* models only capture the status of a graph at a point of time. Therefore, *incremental* dynamic graph analytics models have an advantage over the 'resolution' of analytics; however, designing graph data store that is supporting such workloads is challenging. To enable high performance incremental graph analytics, graph stores have to provide high performance graph updation without sacrificing performance of graph analytics.

For example, genomic sequence assembly leveraging de Bruijn graphs [35] [95] incrementally build graphs during sequence assembly. These applications construct a graph based on relatively short fragments of DNA from a sequencer. When a new sequence is generated, graph analysis and graph construction/updates are conducted simultaneously. Therefore, not only is high performance dynamic graph construction required, but also efficient data retrieval during graph analysis.

4.1.2 Design Principle

As described in 2.3, although static graph data structure such as adjacency-matrix and Compressed Sparse Row (CSR) can provide high data locality, re-constructing a graph with the formats take too long time to catch up with given requirements when the topology of the graph frequently changes.

In addition, we assume that graph processing frameworks that use our data store for its underling graph storage follows a think like a vertex programming model, referred to vertex-centric model, likewise many graph processing systems [58, 57]. On graph processing frameworks that employ vertex-centric model, users express actual computation for a single vertex assuming each vertex has information of all its adjacent edges; the frameworks run graph analytics workloads by applying the given computation to each vertex.

For the above reasons, our DegAwareRHH follows an adjacency-list data structure format, whose components are a vertex-table and edge-lists. A vertex-table holds a set of all vertices. Each element of a vertex-table consists of a pointer to an edge-list and also vertex property data if needed. An edge-list holds a list of adjacent edges of a vertex and edge property data.

4.1.3 Underling Data Structure

To implement an adjacency-list format, several types of data structures can be used (Table 4.1). A vector (one-dimensional array) is the most simple and compact data structure, and any operations to a vector result in sequential accesses to a contiguous memory region; however, as there is no function to locate a specific element stored in the data structure, the theoretical worst-case time of lookup operation result in $O(n)$.

A tree is one of the most common data structure used to store large number of elements. Although there are many different ways to represent trees, the theoretical worst-case time of common hash table operations (insert, delete, lookup) of trees can be brought down to $O(logn)$ rather than $O(n)$.

A hash table is another approach to store large number of elements with indexes. In general, there are three strategies to resolve hash collisions: *separate chaining*,

Table 4.1: Candidates for Underling Data Structure

Data Structure			Indexes	Sequential Access
Vector (1-dimensional array)				✓
Tree			✓	
Hash table	Separate chaining	Linked chaining	✓	
	Open addressing	Linear probing	✓	✓
		Quadratic probing	✓	
		Double hashing	✓	

open addressing and *double hashing.* In order to reduce pointer accesses and improve sequential accesses, we use an open addressing and linear probing hash table to implement our dynamic graph data store. The detail of our hash table is described in the next section.

4.2 Robin Hood hashing (RHH)

In this section, we describe a hash table which is used in DegAwareRHH. In order to minimize not only the number of cache misses but also the number of accesses to NVRAM, resulting in page misses, we choose Robin Hood hashing [27], because of its locality properties and compactness. Robin Hood hashing is designed to maintain a small variance of probe distances as well as average probe distance. A probe distance is the distance between initial (hashed) position and current position of a key, and it is an important factor to determine the performance of a hash table. Previous work has actually shown that probe distance can remain small under various load conditions [64, 37]. We expect that the combination of low probe distance and sequential memory accesses patterns of Robin Hood hashing will be beneficial for out-of-core processing using NVRAM.

Cuckoo hashing [68] is another open addressing hash table that resolve hash collisions of elements in a table. Different from Robin Hood hashing, Cuckoo hashing uses two hash functions to resolve hash collisions. However, because Cuckoo hashing causes non-contiguous memory accesses, we choose Robin Hood hashing. For the same reason, quadratic probing, which uses successive values of an arbitrary

quadratic polynomial to resolve collisions, also won't fit to our target workload. In this study, we skip the detailed experimental analysis of the existing hash tables. Detailed analysis of these hash tables can be found in [73].

4.2.1 Insert

Now, we describe how Robin Hood hashing behaves during insert, delete, and search operations. The core features and characteristics of the open addressing hash table are how to deal with hash collisions. The main idea of finding an alternative position in case of a collision in Robin Hood hashing is performing linear probing to the cyclized probe sequence $H(k), H(k)+1, ..., L-1, 0, 1, ..., H(k)-1$, where $H(k)$ is hash value of k and L is the length of the table. The length of the probe sequence is called *probe distance*. Specifically, Robin Hood hashing tries to move an existing element if a new element hashes into its position, and the probe distance of the existing element is equal or smaller than that of the new element. This strategy attempts to keep the average probe distance of all elements in the table small.

An illustration of edge insert using Robin Hood hashing is shown in Figure 4.1. Let {i, j} be an edge between a vertex i and j. In this example, a hash function is $h = v_{id}\ mod\ C$ where v_{id} is a ID of a vertex and C is the capacity of a hash table. First, compute a hash value (initial position) of a new edge {1,5} using the hash function. Second, edge {1,6} is moved to the next position since the probe distance of {1,6} is equal to the one of edge {1,5}. Next, edge {2,2} is moved to the next position and edge {1,6} is inserted in the position (index 2) because probe distance of edge {2,2} is smaller than the one of edge {1,6}. Ideally, a target element is located in the same memory page or cache line where its hashed position is located, although the element is moved to another position because of hash conflict.

4.2.2 Delete

When delete an element, instead of simply erasing all data (probe distance, key and value) of the element and moving succeeding elements forward, we make the element as deleted by setting a *tombstone* flag, keeping its existing probe distance. Thus, a new element is not inserted there if the probe distance of the deleted element is less

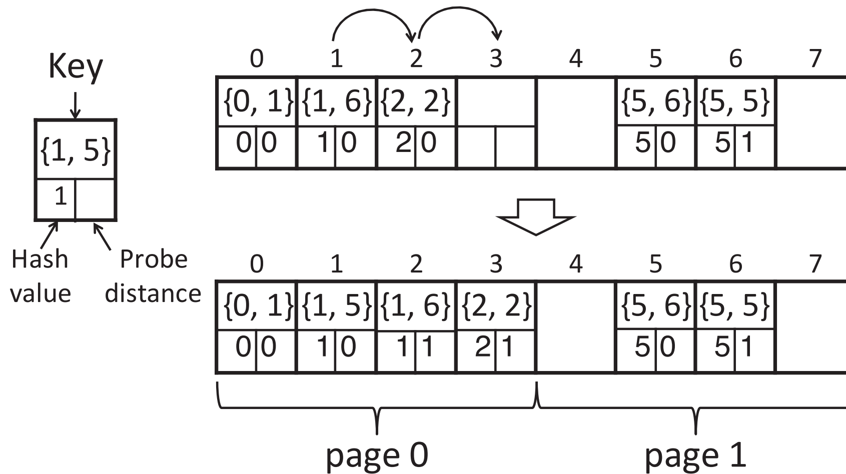


Figure 4.1: Edge insert into Robin Hood hashing table

than the new element. By keeping probe distances of deleted elements, it can be expected to reduce the cost of moving elements when insert or delete an element. The main drawback of this approach occurs after many delete operations have been done. After many elements are deleted, an average prove distance of the table will be large, although the number of actual elements is small. In a worst-case, the probe distance may exceed the capacity of a table. To prevent this situation, we periodically re-hash a whole element of a table so that erase tombstones of deleted elements and reduce probe distances of active elements.

4.2.3 Search

A search operation is straight forward: compute the hash value of a target element and probe the target element linearly from the hash value position. If an empty space (not including tombstones) or an element which has larger probe distance than the target is found, fault the search since the target element does not exist in the table.

4.3 DegAwareRHH

In this section, first, we describe the design of our dynamic graph data store (DegAwareRHH). Second, we describe its insert and deletion algorithms. Last, we

give some optimization techniques that improve its performance.

4.3.1 Overview

DegAwareRHH is designed to target graphs that have following characteristics: 1) extremely-large-scale and scale-free graphs that have at least hundreds of billions–trillions of edges; 2) each vertex and edge may have property data; 3) there may be multiple edges between two vertices, called a multigraph.

For the type of graphs, the key requirements for designing a dynamic graph data store are: 1) high performance inserts and deletes of vertices and edges; 2) quickly locating a specific edge matching topological constraints; 3) providing high locality when reading all adjacent edges of a vertex in order to improve graph analytics performance; 4) supporting out-of-core data structures in order to process large-scale graphs.

To meet the requirements above we propose DegAwareRHH, a high performance dynamic graph data store designed for scaling out to store large scale-free graphs by leveraging the compact and high data locality hash table. DegAwareRHH follow the adjacency-list data structure format using Robin Hood hashing. Additionally, it is designed to manage scale-free graphs by using two different data structures depends on the vertex type. Many real-world graphs can be classified into scale-free graphs where degree distribution follows a power law, that is, most of number of vertices have small number of edges while few vertices have huge number of edges. This degree distribution causes many performance and memory usage problems when processing scale-free graphs. Therefore, in the context of designing a graph data store, adopting different data store models depending on the degree of each vertex is essential.

4.3.2 Data Layout

An illustration of DegAwareRHH is shown in Figure 4.2. Depending on the degree of a source vertex, edges are stored in two types of tables: *low-degree table* and *middle-high-degree table*.

4.3.2.1 Middle-high-degree Table

Locating a specific edge from high degree vertices may be highly costly. Therefore, to quickly locate a specific edge matching topological constraints, we use a hash table for the edge-list instead of using a simple 1-dimensional array. This data structure efficiently handles inserts while maintaining locality of accesses. The *middle-high-degree table* consists of a *vertex table* and *edge-chunks*. The *vertex table* stores source vertices' information, i.e., source vertex's ID and vertex property data if needed. Adjacency edges are stored into *edge-chunks* and the *vertex table* holds pointers to *edge-chunks*. In order to locate a specific edge with a constant time, we also use Robin Hood hashing for *edge-chunks*. In our current implementation, only 1 byte of extra space is allocated for each element to construct a Robin Hood hashing table (7 bits for a probe distance, and 1 bit for a tombstone flag).

4.3.2.2 Low-degree Table

Even though we use the compact hash table, there are some relatively high cost operations for low-degree vertices, such as allocating a new table and pointer accesses to edge-lists, especially in out-of-core situations. Accordingly, we allocate a single Robin Hood hashing table to store low-degree vertices in order to reduce the costs that are relatively high for low-degree vertices. The *low-degree table* stores edges in a single compact table, i.e., directly using Robin Hood hashing.

To be noted, while setting a high threshold for the middle-high-degree will reduce the cost of inserting edges into the middle-high-degree table, it causes many hash conflicts in the low-degree table and will slow down it. Thus, for future work, from the perspective of improving the performance of Robin Hood hashing itself especially with many hash conflicts, techniques such as using bucket [87] maybe be used.

4.3.3 Programming API

Like existing popular graph processing frameworks, DegAwareRHH provides basic APIs including: graph construction and update APIs such as add_edge, delete_edge, and update_property_data; graph reading APIs such as get_property_data and

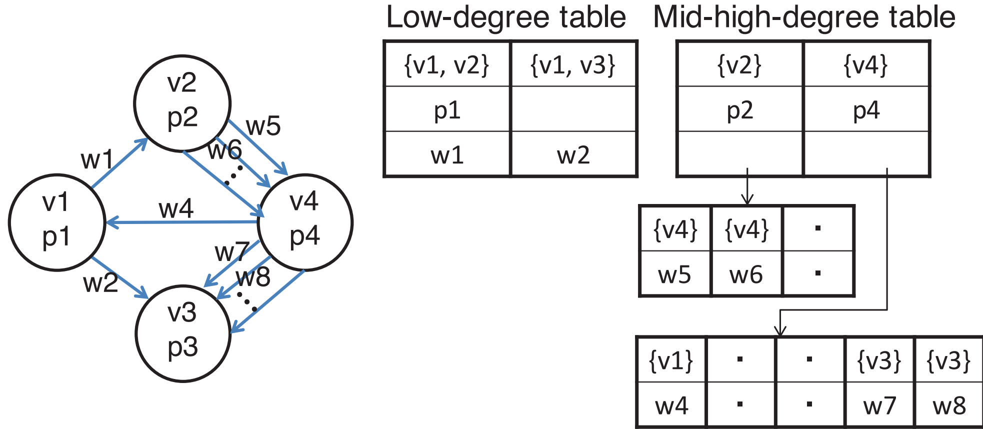


Figure 4.2: Overview of Degree Aware Dynamic Graph Data Structure (DegAwareRHH)

get_adjacent-edges. The get_adjacent-edges function return a forward iterator pointing to an *edge-chunk*. Using C++ templates, our implementation can hold any type of object for key, value, and property data.

4.3.4 Insert Algorithm

An illustration of the edge insert algorithm of DegAwareRHH is shown in Figure 4.3. Let (u, v) be an edge to be inserted where u is a source vertex ID and v is a target vertex ID.

Step 1. Query the current degree (the number of edges) of vertex u to the low-degree table.

Step 2-A. If the degree is more than 0 and less than the low-degree threshold of the middle-high-degree table, insert the edge into the low-degree table. If the degree of the low-degree vertex is equal to the low-degree threshold, move all edges of the vertex to the middle-high-degree table.

Step 2-B. Otherwise, query the current degree of vertex u to the middle-high-degree table. If the degree is 0, that is, vertex u is a new vertex, then insert into the low-degree table. Otherwise insert the edge into the middle-high-degree table.

Step 3. While inserting an element, if the probe distance of an element exceeds

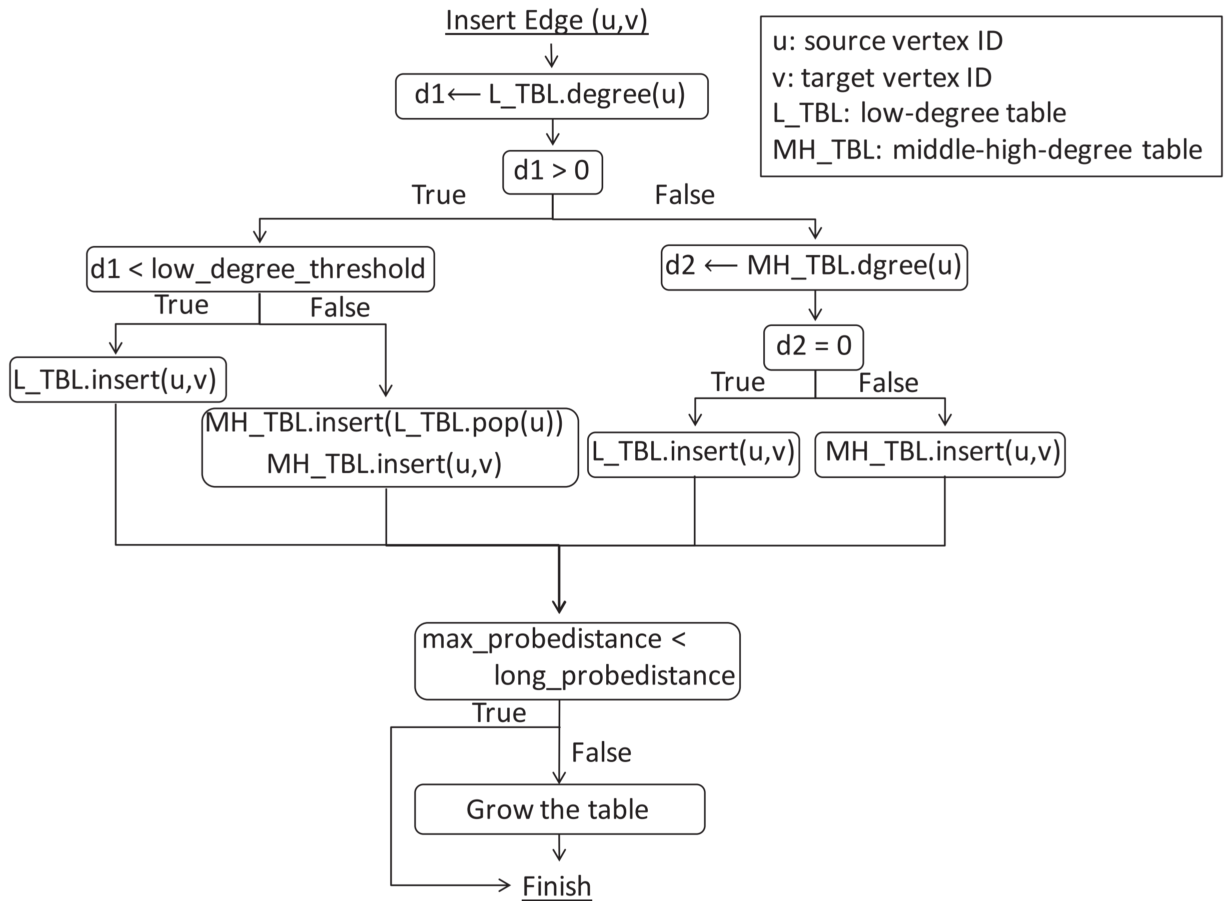


Figure 4.3: Degree Aware Edge Insert Algorithm

the threshold (long_probedistance) due to too many hash conflicts, the table is grown to double in size. It is well known that a probe distance usually keeps small constant number, and it will rarely happen that a probe distance becomes long.

4.3.5 Delete Algorithm

An illustration of an edge delete algorithm of DegAwareRHH is shown in Figure 4.4. Let (u, v) be an edge to to deleted where u is a source vertex ID and v is a target vertex ID.

Step 1. Find the edge from the low-degree table.

Step 2-A. If the edge is found, delete it from the low-degree table.

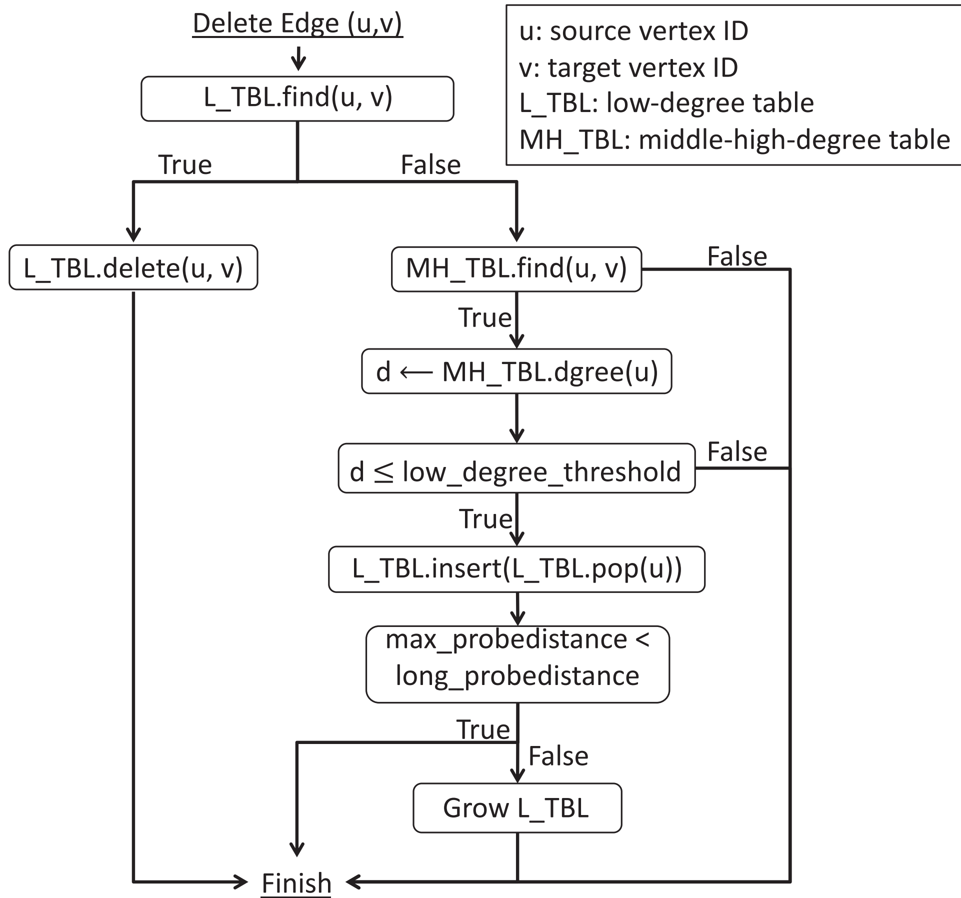


Figure 4.4: Degree Aware Edge Delete Algorithm

Step 2-B. Otherwise, find the edge from the middle-high-degree table. If it is found, delete it from the table and query current degree of vertex u. If the degree become equal or less than the low-degree threshold after deleting the edge, move all edges of the vertex to the low-degree table. While moving the edges to the low-degree table, if the probe distance of an element exceeds the threshold (long_probedistance), the table is grown to double in size.

4.3.6 Optimization

4.3.6.1 Load Factor

It has been reported that when the table is close to full, it degrades its performance rapidly [27]. To prevent this situation, we double the size of the table when the number of elements, not including tombstones, exceeds 90% of its capacity.

4.3.6.2 Rehashing Table

As described in section 4.2, after many elements are deleted and inserted, probe distances may become quite large because we don't reset their probe distances when delete elements to avoid the cost of moving the following elements. Thus, we rehash all elements in a table when the maximum probe distance of the table exceeds the length of the table. In addition, we rehash tables after insert or delete elements over a certain number of times (for example, usually 1–10 millions), contributing to better performance with negligible execution time, especially in out-of-core workloads.

4.3.6.3 Memory Pool Allocator

Although DegAwareRHH stores low degree vertices into the low-degree-table, a lot of relatively small size of memory allocation is performed to store middle degree vertices. To efficiently conduct memory allocation operation, we use a memory pool allocation technique.

4.4 Extending DegAwareRHH for Distributed-Memory

4.4.1 An Asynchronous Distributed-Memory Communication Framework

Distributed-memory DegAwareRHH has been implemented with MPI to store a distributed graph. We adopt the distributed asynchronous visitor queue framework, provided by HavoqGT (Highly Asynchronous Visitor Queue Graph Toolkit) [70, 71], to be the driver of DegAwareRHH. HavoqGT is an open source[1] graph analytics framework that provides infrastructure to develop asynchronous vertex-centric graph algorithms. The visitor queue framework provides the parallelism, and creates a data-driven flow of computation over MPI with asynchronous communication. All visitors are asynchronously transmitted, scheduled, and executed. The framework targets parallel and distributed environments and large scale-free graphs. Pearce et al. demonstrated that this framework provides excellent scalability [70, 71].

Before describing the details of how to extend DegAwareRHH for distributed-memory over the asynchronous visitor queue framework, we point out advantages of employing an asynchronous communication model in contrast to synchronous communication model.

First, a synchronous communication model incurs a global synchronization every super-step, and one of its actual model is the Bulk Synchronous Parallel (BSP) [85] model, which is widely used in many distributed graph processing frameworks such as Pregel [58], Apache Giraph [17], Apache Hama [79], GraphX [91]. Another popular synchronous communication model is GAS (Gather, Apply, and Scatter) model used in PowerGraph [41] and PowerLyra [32]. While synchronous communication models are widely used in many distributed-memory graph processing framework and easy to use, one of the major disadvantages of using synchronous communication models for dynamic graph analytics is that its global communication is relatively heavy when perform a fine-grained graph update, e.g., add only one edge in a graph and update graph analytics results if needed.

[1] https://github.com/LLNL/havoqgt

On the other hand, although asynchronous models are not easy to use and implement, they can perform relatively low cost communications within the area need to update. Thus, we can expect efficient remote communications for fine-grained (incremental) dynamic graph analytics.

4.4.2 DegAwareRHH on the Asynchronous Visitor Queue

In order to extend DegAwareRHH for distributed-memory on the visitor queue framework, we first need to determine an allocation strategy for vertices – i.e., which process is responsible for a given vertex. To determine the owner process of a new edge request e_{id}, consisting of an operation and a *source*, *destination* pair, we use Consistent Hashing [49]. A vertices' owner process is computed as follows: $hash(e_{id}.source)\ mod\ P$, where P is the number of processes, and all processes use the same hash function. By using this strategy, any process can determine, in constant time, the owner of a given vertex. Although these strategies are simple, we believe that using these strategies is enough for evaluating the performance of our graph data store. We would like to explore other graph partitioning strategies as one of the future work.

An illustration of a distributed dynamic graph construction algorithm is shown in Figure 4.5. Note that, in Figure 4.5, we only show the communication from *Process A* to *Process B* to make the figure concise. Each process independently can read graph update requests (insert/delete) and passes them to the visitor queue one by one. The visitor queue applies the request immediately into the local graph store if it is the owner of the source vertex; otherwise, if the owner of the source vertex is a remote process, it pushes the request into its local message queue being wrapped in a visitor object, and send the visitors asynchronously when the queue becomes full. Each process checks its receive queue periodically, and applies received graph update requests to the local store, i.e., it applies the graph construction requests into the local graph data store.

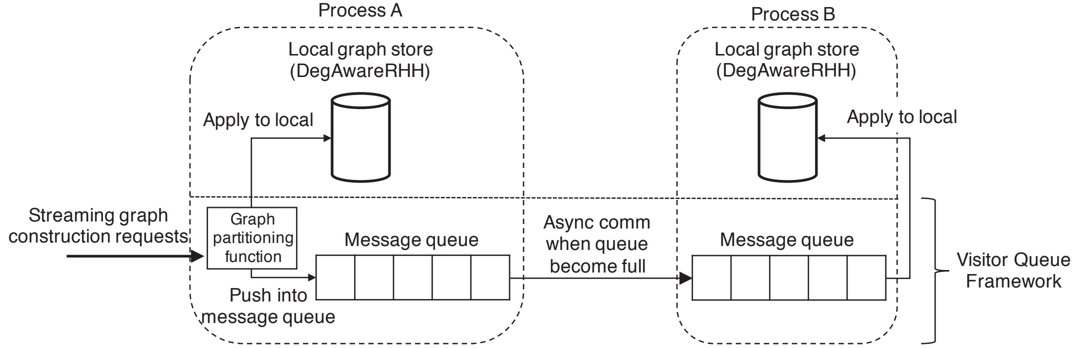


Figure 4.5: Distributed Dynamic Graph Update over the Visitor Queue Framework (note that process B also can stream graph construction requests and send the requests to process A)

4.5 Experimental Setup

In this section, we experimentally evaluate the graph construction performance of our dynamic graph data store (DegAwareRHH). Specifically, we perform experiments with the following 5 scenarios:

- Graph construction
- Static BFS algorithm
- Dynamic graph algorithm
- Out-of-core graph construction
- Out-of-core graph construction on future NVRAM

4.5.1 Details of Implementation of DegAwareRHH

We implemented DegAwareRHH in C++ and used the *Boost.Interprocess* library to allocate in a memory mapped region. Specifically, we use the *Boost.Interprocess* memory pool allocator to allocate the hash tables to reduce the cost of many small size memory allocations. We use *mmap* system call as an interface to the NVRAM. For in-core experiments, we create files on tmpfs so that explicitly allocate graph data on DRAM.

Table 4.2: The Size in Bytes of Each Element in Tables in DegAwareRHH

Table	Size	Breakdown
low-degree	19 B	1 B for management (probe distance and delete flag) 16 B for an edge (source vertex ID and target vertex ID) 2 B for property data of the source vertex and edge
middle-high-degree	18 B	1 B for management and 8 B for a source vertex ID 1 B for property data of the source vertex ID 8 B for a pointer pointing to its edge-chunk
edge-chunk	10 B	1 B for management and 8 B for a target vertex ID 1 B for property data (edge)

Before beginning the series of experiments, we first performed a preliminary experiment to determine middle-high-degree threshold. We used a single compute node of Catalyst with 24 processes; constructed a RMAT scale 26 graph on its local NVRAM. Setting middle-high-degree threshold to 2, approximately 7% of speed up was obtained compared with a case only using middle-high-degree table. Based on the result, we set middle-high-degree threshold at 2, that is, vertices with 2 or more edges are stored in the middle-high-degree table.

The size of each element in the tables in DegAwareRHH when 8 bytes is used to represent vertex ID and 1 byte is for property data are shown in Table 4.2.

4.5.2 Implementations for Comparison

For experimental comparison, we show the performance of the following implementations:

- BaselineVec – BaselineVec consists of a *vertex table* and *edge tables*: *Vertex table* holds source vertices' ID, vertices' property data, and pointers pointing to *edge table*s using *Boost unordered_multimap* container; *Edge table* holds target vertices' ID and edges' property data using *Boost vector* container. *Boost unordered_multimap* consists of multiple *buckets* where each one can have any number of *elements*. Elements with the same hash value are chained to the same bucket. Elements chained to the same bucket are connecting by pointers, thus accessing a next element results in a non-continuous memory access.

  When delete an element in a *edge table*, instead of using *Boost vector*'s erase function naively, we delete the element by swapping with the last element in the

edge table to avoid moving succeeding elements forward after each deletion. In *Boost unordered_multimap*, the bucket where an element is assigned is computed by a hash function, and linear search is conducted to chained elements. When a request of insert of an edge come, a linear search is conducted to find the same edge or an empty space to insert the edge if the same edge is not existed.

- BaselineMap – BaselineMap is another baseline implementation for comparison. BaselineMap adapts *Boost unordered_multimap* container to construct a *edge table* instead of *Boost vector* used in BaselineVec in order to avoid linear edge search costs. Meanwhile, the cost of inserting an edge will increase slightly due to a fundamental overhead of using a hash table instead of using a vector; the cost of reading adjacent edges will also increase due to pointer accesses to read chained elements in *Boost unordered_multimap*.

- STINGER [39] – STINGER is a shared memory (in-core) parallel dynamic graph processing framework developed at the Georgia Institute of Technology. STINGER can update a graph with several times–three orders of magnitude better performance in comparison with state-of-the-art 12 open source graph databases and libraries [61], including *SQLite* [66], *Neo4j* [5], *Giraph* [17], *DEX* [60] and *Boost Graph Library* [10]. We used version 06.15.

4.5.3 Datasets

We conduct experiments using both a synthetic graph generator and real-world graph datasets including a real dynamic graph dataset. Those datasets are stored in files as a pair of source and vertex IDs.

4.5.3.1 Static Graphs

We first present a synthetic graph model and 3 real-world graphs in detail that have been used in many large-scale graph processing studies (Table4.3).

- RMAT – Generates scale-free graphs using R-MATA graph generator model [28], and we follow the Graph500 V1.2 specifications for generator parameters [3]. After graph generation, all vertex labels are uniformly permuted to destroy any

locality artifacts from the generators. The graph generator generates a graph as an undirected graph in random edge order. For an edge (e_i, e_j), we also generated the opposite direction of edge (e_j, e_i). RMAT graphs have a 16x edge factor, thus, finally, $vertices \times 32$ directed edges were generated for each RMAT graph. To evaluate the performance of delete operations, we add 5%–40% of edge deletion requests into edge insertion requests with random order, but with a carefully chosen position such that the insertion request of an edge comes before its deletion request.

- Twitter [52] – This graph contains 41,652,230 vertices (user accounts) and 1,468365,182 directed edges. The 42 million users are connected to each other by 2.9 billion follow (follower/following) links. This dataset is generated from the snapshot of the Twitter network topology as of 2009.
- SK2005 (hyperlink graph) [23] – This graph contains 50,636,154 vertices and 1,949,412,601 directed edges and has been obtained from a crawling of the .sk domain in 2005. Each vertex corresponds to a web page and each edge is a hyperlink.
- Webgraph2012 (hyperlink graph) [25] – We also use an extremely large web graph, the largest open source real graph dataset to our knowledge, that has 128 billion edges (as a directed graph). Each vertex corresponds to a web page and each edge is a hyperlink.

Since the datasets do not have any data on vertex and edge, we set vertex and edge property data as a dummy value (NULL). Each vertex is represented as a 64-bit integer value; thus, an edge is a pair of 64-bit integers.

4.5.3.2 Real Dynamic Graph

In addition to the static datasets, we use the largest dynamic graph data set, to our knowledge, from over a decade of the English Wikipedia corpus in order to perform dynamic graph construction experiments at scale with real-world data [75][2]. The graph's vertices are pages and edges are hyperlinks. General statistics about the Wikipedia dataset are showed in Table4.4. This dataset was created by starting with the full historical XML dumps provided by Wikipedia, and parsing the wiki markup

[2] Available at http://software.llnl.gov/havoqgt/datasets.html

Table 4.3: Static Graph Datasets Used in Experiments

Name	#Vertices	#Edges	OnDiskSpace
Twitter	41,652,230	1,468365,182	49 GB
SK2005	50,636,059	3,860,585,896	65 GB
Webgraph2012	3,563,602,686	128,736,914,167	5.1 TB
RMAT($SCALE$)	$2^{(SCALE)}$	$2^{(SCALE)} * 32$	

Table 4.4: Properties of Wikipedia Dataset [75]. *Unique* defines edges/vertices that are ever possibly in existence.

Time Range	Jan 2001 - Dec 2015
# Edge Inserts	2,713,888,893
# Edge Deletes	1,806,190,225
# Unique Vertices	205,774,846
# Unique Edges	1,303,659,380

to extract internal (intra-wiki) and external hyperlinks. This process extracted of the hyperlinks from all historical revisions from all English pages. Consecutive revisions of a page are compared to check if links were created or deleted. All hyperlink creation and deletion events from all pages were captured and sorted by time to create single stream of historical hyperlink records.

4.5.4 Machine Configurations

We use a compute cluster, the Catalyst cluster at Lawrence Livermore National Laboratory, and a single-node machine, Roma. A single compute node of Catalyst has 12-core Intel(R) Xeon(R) E5-2695v2 processors (2 sockets) and 128 GB of DRAM. The compute nodes are connected over dual rail QDR-80 Intel True Scale Fabric. The Roma machine has 20-core Intel(R) Xeon(R) E5-2650v3 processors (2 sockets) and 256 GB of DRAM and is equipped with 4 node-local NVMe 800 GB of NAND flash NVRAM.

To perform a fair comparison, we compiled the implementations using GCC-4.8 in Roma and GCC-4.9 in Catalyst with -O3 optimization option. For BaselineVec, BaselineMap, and DegAwareRHH, we used Boost 1.60.0 and MVAPICH2.

4.6 Experiment 1: Large-scale Dynamic Graph Construction

4.6.1 Experiment Method

In this section, we evaluate DegAwareRHH in the context of graph construction performance on a single-node and multi-nodes. We repeated the following steps:

- Step 1) each process buffers a subset of edges (1 million) from disk into DRAM to avoid measuring the cost of reading edges from files.
- Step 2) insert or delete edges from the edge buffer into the graph data store sequentially. When insert an edge uniquely, an actual insertion operation is performed following a *find* operation.

We only measured the execution time of step 2.

4.6.2 Single Process Experiments

4.6.2.1 Graph Construction Performance

Before beginning the series of experiments, we first evaluate graph construction performance of BaselineVec, BaselineMap, and DegAwareRHH without HavoqGT visitor queue framework. We use Roma machine (single process) and RMAT SCALE 25 graph (1 billion edge insertions) with 5% of additional edge deletions. The results are shown in Figure 4.6. The x-axis indicates the number of processed edge update

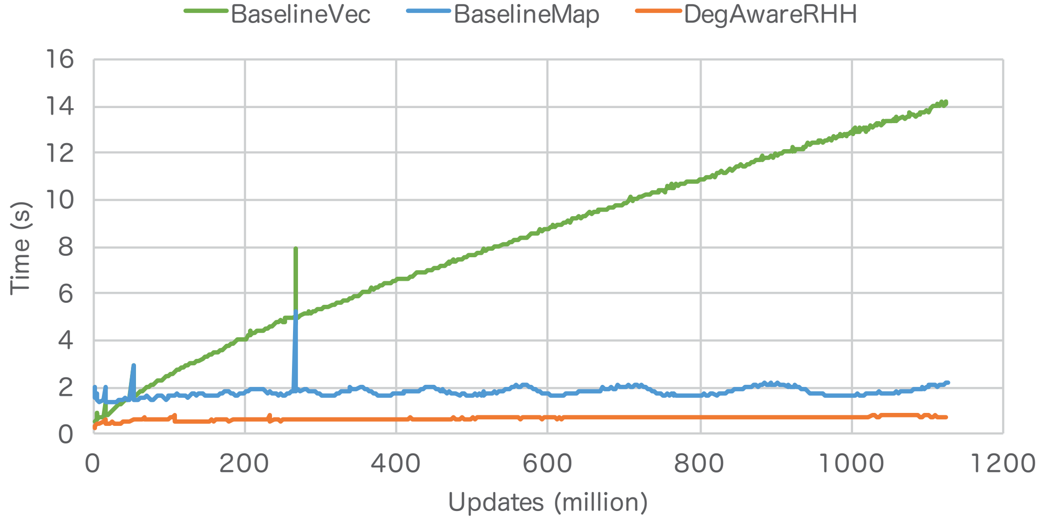


Figure 4.6: Single Process Unique Edge Insertion and Deletion (RMAT SCALE 25, in-core, single-process)

requests (insert/delete) in million and the y-axis indicates the execution time of processing 1 million requests in seconds. As the number of processed updates increases, the performance of BaselineVec decreases due to increasing the size of its edge-list tables. On the other hand, BaselineMap and DegAwareRHH results in constant execution time. DegAwareRHH can insert 1 billion edges and delete 5% of additional edges in 773.3 seconds and outperforms BaselineVec by 11.9x and BaselineMap by 2.6x.

The differences of the results come from differences of the data structures. The BaselineVec stores edges into a vector container, and it requires sequential access to find a target edge; thus, BaselineVec slows down as the size of a constructed graph increases. On the other hand, as BaselineMap and DegAwareRHH use hash tables for edge tables, the execution time can be near constant even though the number of inserted edges increases.

4.6.2.2 Analysis of Probe Distance

To confirm that our implementation of Robin Hood hashing can keep low probe distance when constructs a graph, we monitored how the average probe distance of the tables in DegAwareRHH, i.e., low-degree table, middle-high-degree table and edge-

chunks. In this experiment, tables were not re-hashed after inserting/deleting a certain number of edges to monitor the worst case.

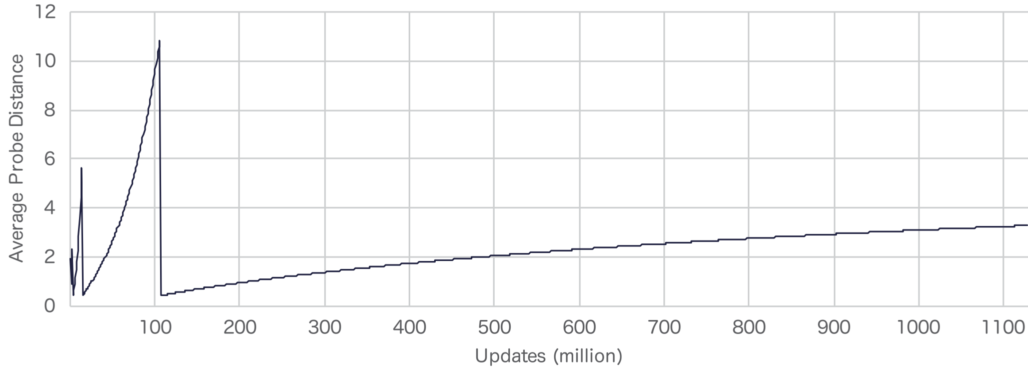
The variations of the average probe distances of the low-degree table and middle-high degree table with the graph construction experiment are shown in Figure 4.7. Surprisingly, the average probe distances of the two tables remain low, specifically that is under 11 even at most. The points where values are suddenly dropping correspond to the time when the tables' capacity (length) are changed.

The distribution of the average probe distances of the edge-chunks after finishing the graph construction is shown in Figure 4.8. The y-axis denotes the number of edge-chunks on a logarithmic scale whose average probe distances are in the range of values on the x-axis. Only a single edge-chunk's average probe distance fell within the two ranges, $[11, 12)$ and $[12, 13)$, respectively; there is no edge-chunk whose average probe distance is greater than or equal to 13. The average probe distances of 86.05 % of edge-chunks are less than 1.

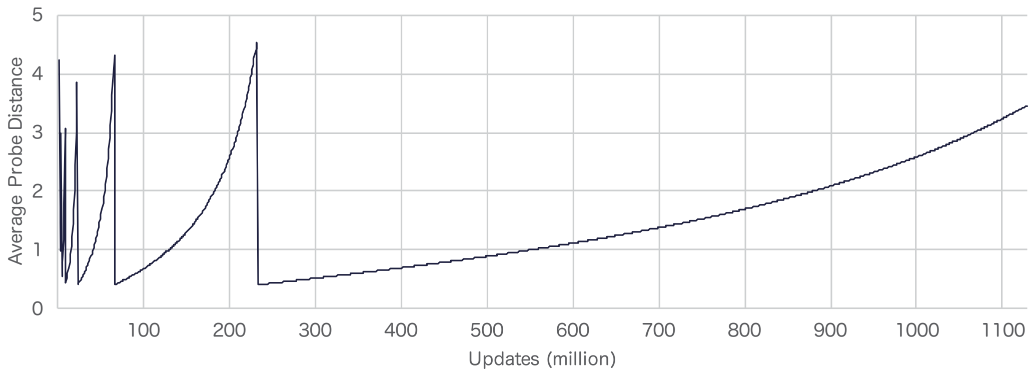
These results indicate that Robin Hood hashing works well under such graph construction workloads.

4.6.3 Single Node Experiments (vs STINGER)

Next, we evaluate graph construction performance of BaselineVec, BaselineMap and DegAwareRHH against STINGER on a single node with multiple processes. This experiment also aims at confirming the validity of performances of BaselineVec and BaselineMap to perform the rest of experiments. We show graph construction performance using a RMAT SCALE 25 graph and additional 5% of edge deletion with the implementations in Figure 4.9. We measure performance for 6, 12, and 24 threads (STINGER) or processes (BaselineVec, BaselineMap and DegAwareRHH). The x-axis denotes the implementations with varying number of threads/processes. The y-axis denotes the speed up against STINGER with the same number of threads/processes. All implementations strongly scale with increasing the number of threads/processes from 6 to 24, by 2.6 times on STINGER, by 1.5 times on BaselineVec, by 3.6 times on BaselineMap, and by 3.5 times on DegAwareRHH. BaselineVec, BaselineMap, and DegAwareRHH outperform STINGER, by 5.6 times, by101.2 times, and by 212.2 times with 24 threads/processes, respectively.



(a) Low-Degree Table



(b) Middle-High-Degree Table

Figure 4.7: The Variations of Average Probe Distance When Constructing a Graph

STINGER stores a graph using adjacency-list model same as the other implementations; however, it stores edges into a linked-block-list (each linked-block has multiple edges) data structure, and it require a sequential search to find a target edge or an empty space for a new edge. Thus, the time to insert an edge (i, j) increases as the out-degree of i increases. Different from STINGER, BaselineVec always inserts a new edge into the last position of the array. Due to this, BaselineVec performs better than STINGER.

Second, we evaluate graph construction performance varying the number of edge deletion requests on the four implementations with 24 threads/processes. The results are shown in Figure 4.10. We changed the rate of added edge deletion requests as 5, 10, 20 and 40. The x-axis denotes the rate of deletion requests, and the y-axis denotes the number of processed graph construction requests per second in log scale. As the number of deletion requests increases, the performance of STINGER and BaselineVec increases by 65% on STINGER and by 44% on BaselineVec since the

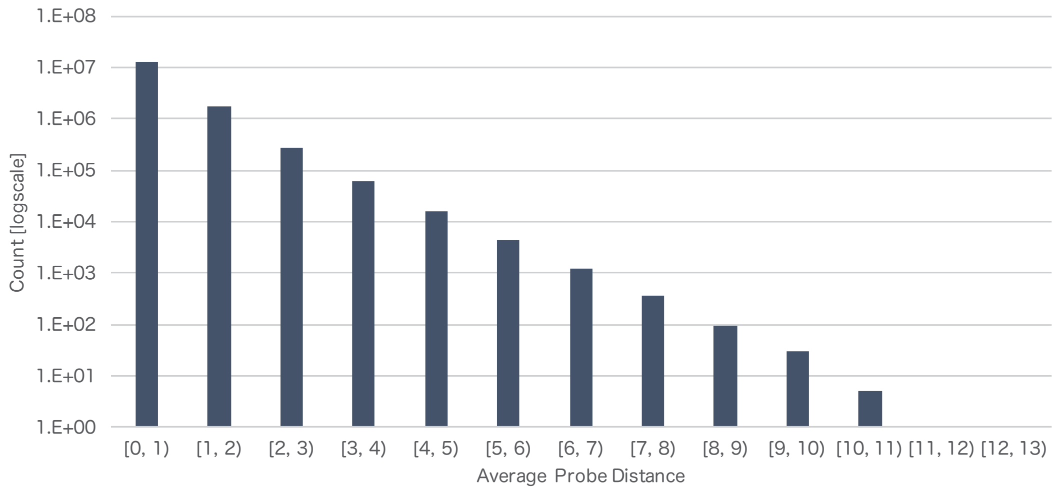


Figure 4.8: The Distribution of the Average Probe Distances of the Edge-Chunks

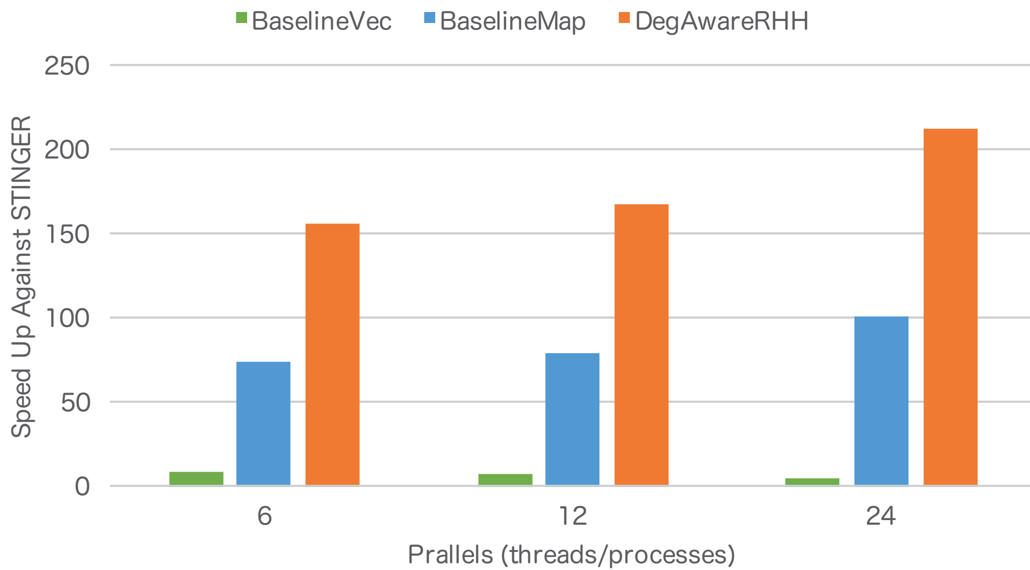


Figure 4.9: Speed Up for Unique Edge Insertion and Deletion Against STINGER (RMAT 25, in-core, single-node). Higher is better.

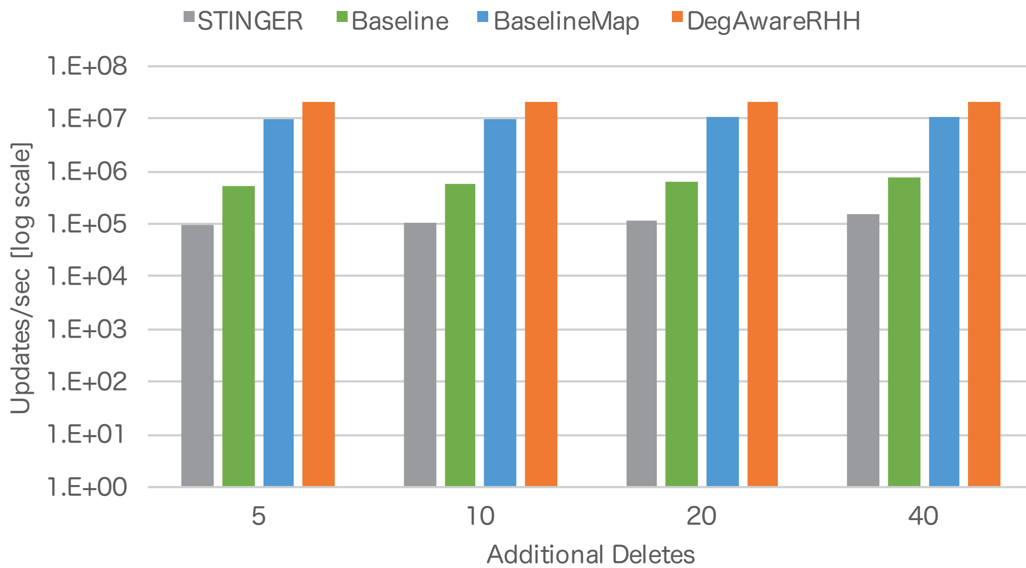


Figure 4.10: Single-node Unique Edge Insertion and Deletion (RMAT 25, in-core, single-node). Higher is better; note log y-axis scale.

cost of finding an edge decreases because the size of edge tables get smaller. In contrast, BaselineMap and DegAwareRHH speed up a little, by 10% on BaselineMap and by 5% on DegAwareRHH, as the number of deletion requests increase. Nevertheless, DegAwareRHH still outperforms the other implementations, e.g., 134x faster than STINGER, 28x faster than BaselineVec, and 2x faster than BaselineMap at the 40% of additional edge deletion requests case.

4.6.4 Multi-Node Experiments

In this section, we evaluate graph construction performance of DegAwareRHH against the BaselineVec and BaselineMap on multiple nodes. Note that we don't use STINGER in this experiment, since it only supports a shared memory environment.

Weak Scaling To explore how DegAwareRHH scales when multiple processes run on multiple compute nodes, we first perform a weak scaling experiment which increases the number of compute nodes while fixing the size of the sub-graph per node. Each compute node constructs a subgraph which has 34 million vertices and over 537 million undirected edges, thus, the actual number of inserted edges on a compute node is over

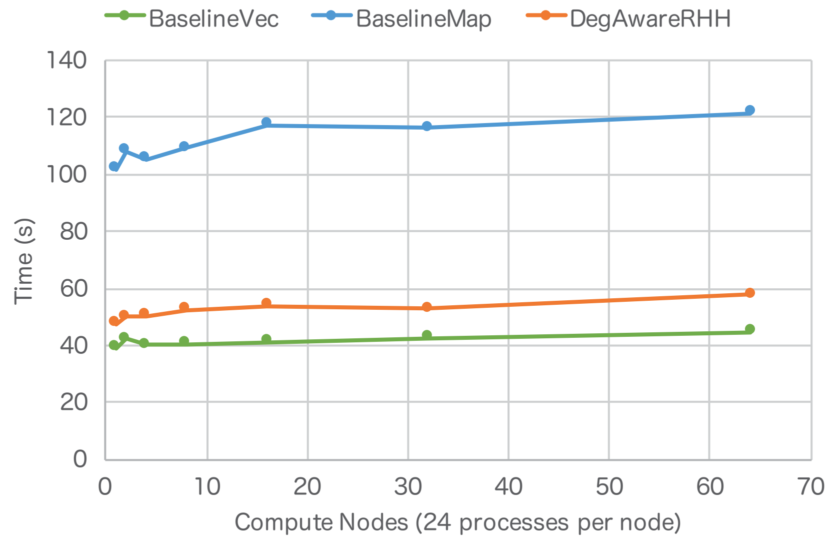
1 billion. We also use datasets which has 5% of additional deletes. We increase the number of compute nodes up to 64 with 24 processes per node, therefore, at 64 nodes, the graph that has 2 billion vertices and 69 billion edges are inserted.

As described before, DegAwareRHH is designed to find a specific edge quickly by using hash tables of its edge-chunk. To evaluate the overhead of inserting edges into the hash tables instead of simply inserting into arrays like BaselineVec does, we first perform non-unique edge insertion experiments, i.e., insert an edge without checking whether the edge is already existing.

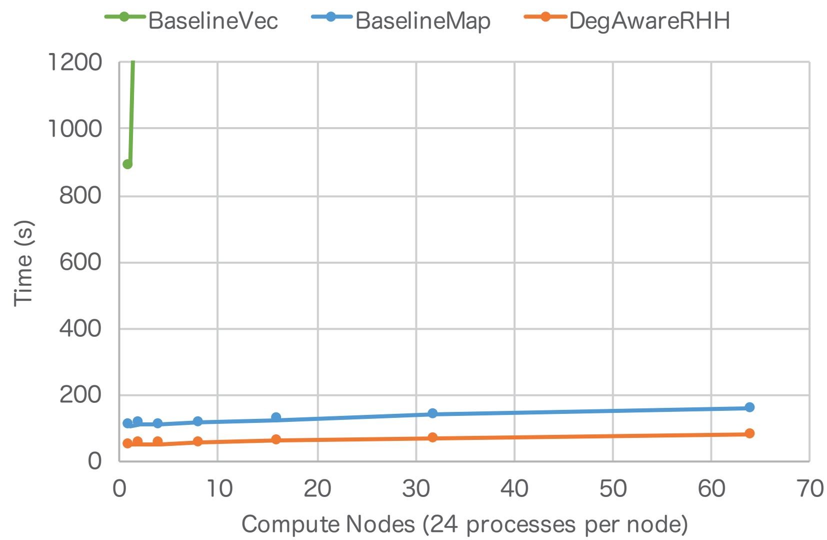
The top figure in Figure 4.11 shows the execution time of the non-unique edge insertion workload. As the number of nodes increase, the three implementations scale; thus, first it is confirmed that our choice of HavoqGT as the underling distributed-memory communication framework can scale with enough performance. Second, as expected, BaselineMap and DegAwareRHH show lower performance than BaselineVec. At 64 nodes with 1536 processes, BaselineVec takes 44.6 seconds, BaselineMap takes 121.4 seconds, and DegAwareRHH takes 57.7 seconds to insert 69 billion edges. This result can be attributed to the edge insertion algorithm of BaselineVec that simply inserts an edge at the last position of an edge-chunk of which underling data structure is a vector. However, as a point of interest, the performance degradation, compared with BaselineVec, on DegAwareRHH is only less than 30% while that of BaselineMap is over 250%. Thus, it is confirmed that the design of the hash table used in DegAwareRHH can insert elements with reasonable low overhead.

The bottom figure in Figure 4.11 shows the execution time of the non-unique edge insertion workload with 5% of additional deletes. At 64 nodes and 1536 processes, BaselineMap takes 156.6 seconds and DegAwareRHH takes 80.1373 seconds. We halted the experiments on BaselineVec at more than 8 compute nodes before finishing due to excessive run times. Since BaselineVec have to perform a linear search within an edge-chunk to find a target vertex ID when delete an edge, its execution time on the workload gets worse. Note that the length of edge-chunks for high degree vertices increase drastically due to the skewness of scale-scree graphs with increasing graph's size.

Figure 4.12 shows the execution time of the unique edge insertion workloads with and without edge deletion. Same as the previous experiment, BaselineVec does not scale well and its performance rapidly deteriorates beyond a number of machines. At 64 compute nodes, BaselineVec inserts 69 billion edges with 124.1 seconds and



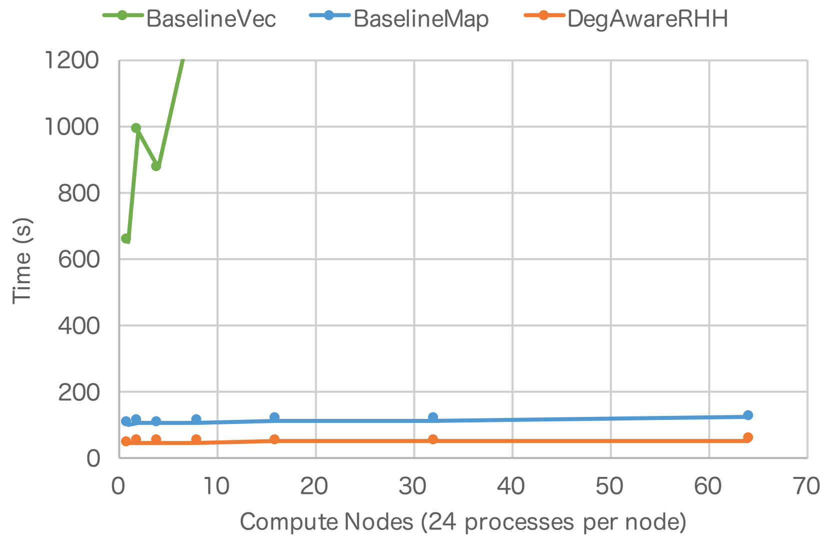
(a) Insertion only



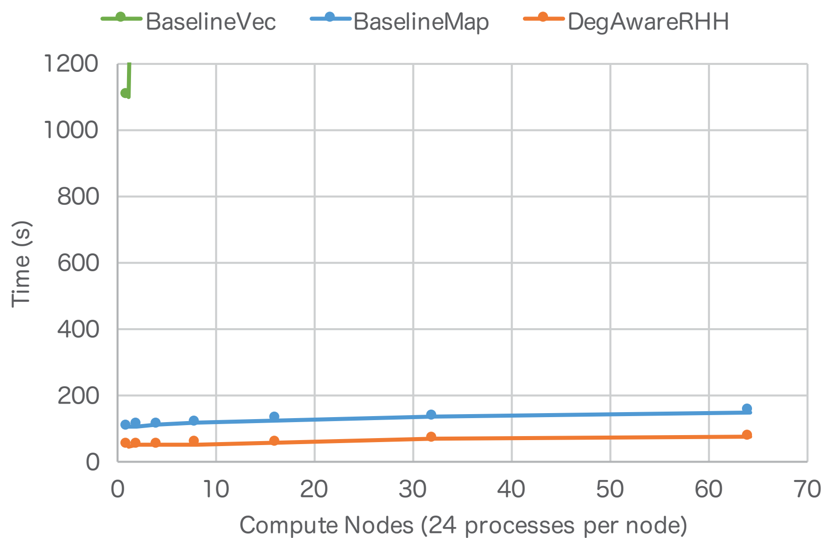
(b) 5% of additional deletion

Figure 4.11: Multi-node Non-unique Edge Updates (24 processes per node, 537 million edges per node)

DegAwareRHH insert the edges with 55.4 seconds. With 5% of additional edge deletion requests and 64 compute nodes, BaselineMap takes 151.0 seconds and DegAwareRHH takes 76.1 seconds.



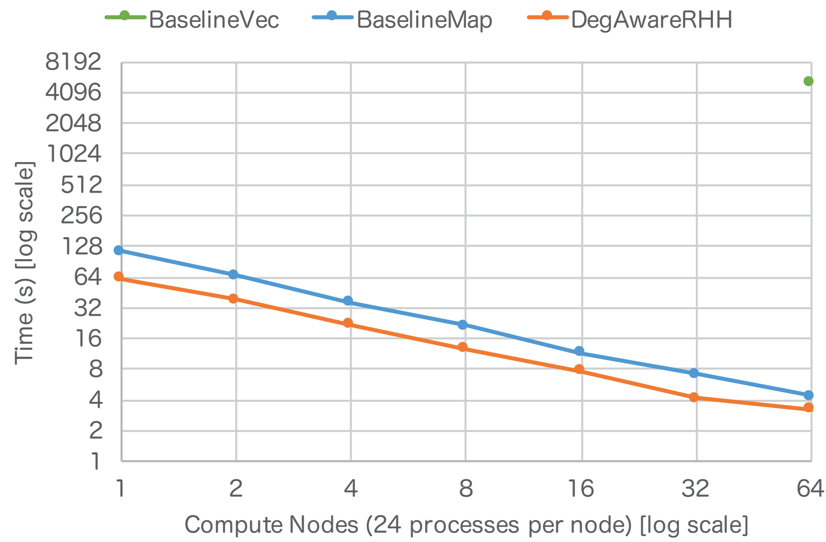
(a) Insertion only



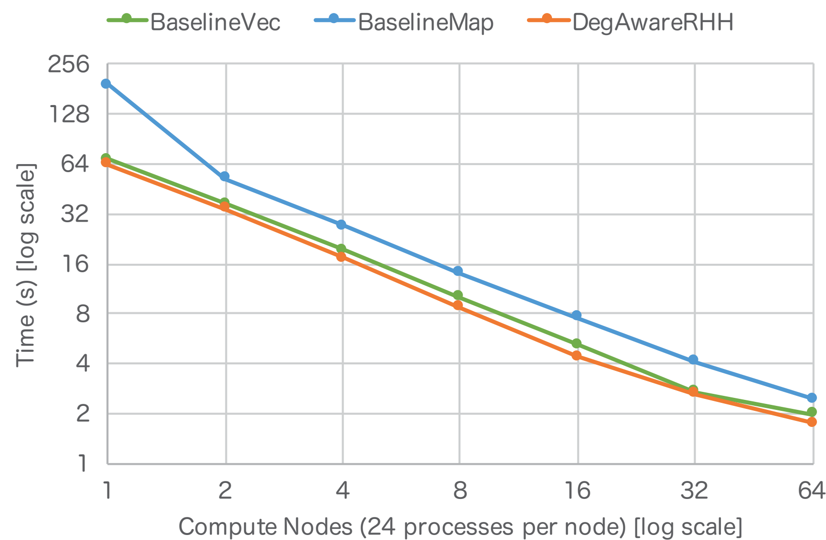
(b) 5% of additional deletion

Figure 4.12: Multi-node Unique Edge Updates (24 processes per node, 537 million edges per node)

Strong Scaling on Real Graphs We perform graph construction experiments on the three real graphs including the massive-scale graph, Twitter, SK2005 and Webgraph2012. For Twitter and SK2005 graph, we increase the number of compute nodes from 1 to 64. For Webgraph2012, BaselineVec and DegAwareRHH can hold the graph on 32 compute nodes, and that of BaselineMap is 128 compute nodes at a minimum due to its large graph size. We increased the number of compute nodes



(a) Twitter



(b) SK2005

Figure 4.13: Muti-node Unique Edge Insertion on Real Graphs (24 processes per node, strong scaling)

from 32 to 192 in increments of 32.

The results of strong scaling experiments on Twitter and SK2005 graphs are shown in Figure 4.13. Because the maximum degree of SK2005 graph is approximately only 12K while that of Twitter graph is over 1M, BaselineVec shows better performance than BaselineMap on SK2005 graph. DegAwareRHH outperforms BaselineVec and

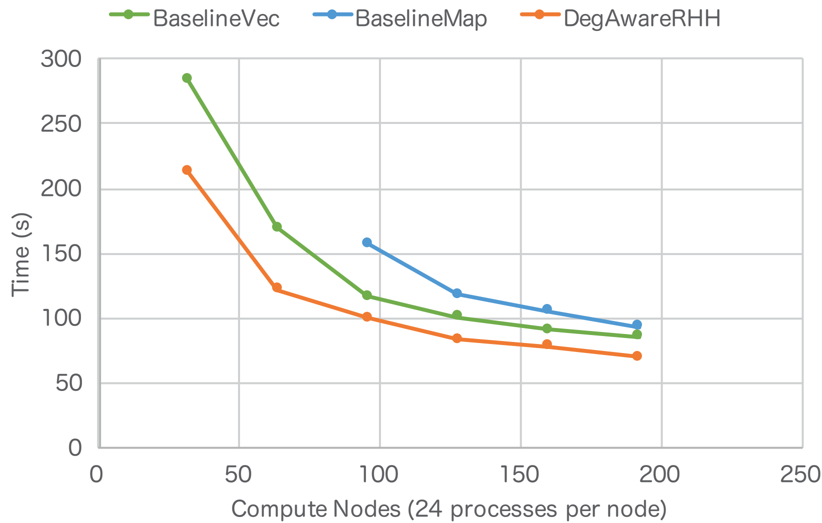


Figure 4.14: Multi-node Unique Edge Insertion on Webgraph2012 (24 processes per node, strong scaling)

BaselineMap on both graphs.

On Webgraph2012, the implementations scale with increasing the number of nodes (Figure 4.14). As the number of high degree vertices in the graph is smaller than that of the RMAT graphs, the performance gaps between BaselineVec and DegAwareRHH shrink; BaselineVec shows better performance than BaselineMap. The max out-degree of the Webgraph2012 is only 35K although its max in-degree is 95M. Nevertheless, DegAwareRHH outperforms BaselineVec by 1.22x and BaselineMap by 1.34x at 196 compute nodes.

4.6.5 Realistic Workload

To evaluate a realistic dynamic graph update workload, we conduct an experiment on the Wikipedia graph dataset. Although the Wikipedia graph is the largest open source realistic dynamic graph datasets in our knowledge, its actual data size is relatively small; thus, we perform this experiment on a single compute node of Catalyst with 24 processes.

Some applications of dynamic graphs may involve multiple parallel streams of edge operations. Thus, to evaluate the performance of DegAwareRHH when

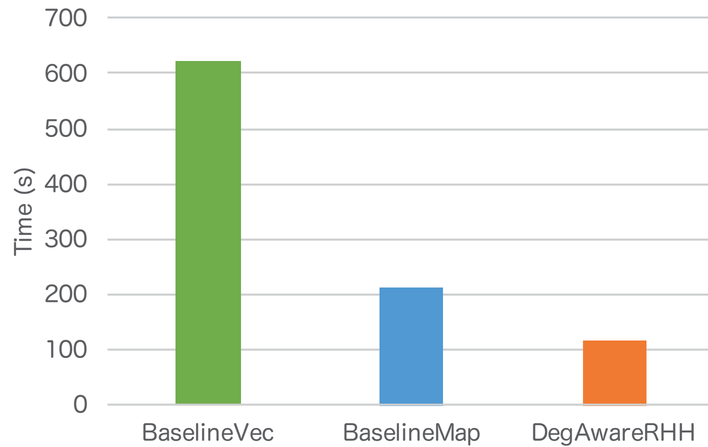


Figure 4.15: Unique Edge Insertion on the Wikipedia Graph

processing parallel streams, we split the single Wikipedia edge stream into multiple parallel streams. Note that our parallel streams will not preserve the exact ordering of the single stream, but serve as a realistic test for performance evaluation.

The execution time to construct the Wikipedia graph on the three implementations is shown in Figure 4.15. DegAwareRHH can process 2.7 billion edge inserts and 1.8 billion edge deletes with 115.50 seconds, and outperforms BaselineVec and BaselineMap by 5.4x and 1.8x, respectively. This result indicates that DegAwareRHH also can provide high performances on realistic dynamic graph update workloads. Similar to the Webgraph, because this graph also has low skewness – max out-degree is small, the performance gap between BaselineVec and the other two implementations are relatively small.

4.7 Experiment 2: Performance Evaluations on Graph Algorithms

In this section, we evaluate the performance of our DegAwareRHH on static Breadth-first search (BFS) and PageRank algorithms.

4.7.1 Experiment Method

4.7.1.1 Implementation

Because the performance of multi-node BFS and PageRank are strongly affected by their design and implementation of remote node communication, we implemented a single process BFS and PageRank algorithms without using the visitor queue program in HavoqGT for this experiment. For performance comparison, we implemented a static graph data store with a CSR data structure.

BFS The pseudo code of a BFS algorithm we use is shown in Figure 4.16. *frontier_queue* and *next_queue* are queue data structures to store the current level's frontier vertices and next level's frontier vertices, respectively. *visited* is a bitmap used to represent whether which vertices are already visited or not. The execution time of BFS is sum of the BFS main loop and a step for initializing *visited*.

PageRank Also, the pseudo code of a PageRank algorithm we use is shown in Figure 4.17. We store each vertices' PageRank value into graph data stores as their vertex property data. In the algorithm, at each super-step, we propagate each vertex's PageRank value to its adjacent vertices through out-going edges instead of using a 'pull' type communication model, that is, each vertex get PageRank values through in-coming edges. We report the average execution time of a single super-step by running it 10 loops. In the PageRank algorithm, the performance of accessing vertexs property data is required in addition to the performance of accessing the adjacent edges of a vertex.

4.7.2 Results

We used Roma machine and constructed a RMAT SCALE 26 graph in /dev/shm to perform in-core experiments. The process of constructing a graph is not timed.

```
1  /// --- BFS Main Loop -------- ///
2  while (true) {
3    /// Loop over the current frontier (level)
4    while (!frontier_queue.empty()) {
5      uint64_t src = frontier_queue.front();
6      frontier_queue.pop();
7      /// Push adjacent vertices to the next queue
8      for (edge_iterator edge = graphstore.adjacent_edges_begin(src);
9           edge != graphstore.adjacent_edges_end(src);
10          ++edge) {
11       uint64 target = edge.target_vertex();
12       bool is_visited = get_bit_flag(visited, target);
13       if (!is_visited) {
14         next_queue.push(target);
15         set_bit_flag(visited, target);
16       }
17     }
18   }  /// end of loop for the current frontier
19   if (next_queue.empty()) break; /// termination condition
20   frontier_queue.swap(next_queue);
21 } /// End of BFS loop
```

Figure 4.16: Single Process BFS Algorithm

```
1  /// --- Propagate each vertex's PageRank value to its adjacent vertices --- ///
2  for (vertex_iterator vertex = graphstore.vertices_begin();
3       vertex != graphstore.vertices_end();
4       ++vertex) {
5    double pagerank = vertex.property_data().pagerank;
6    size_t degree = graphstore.degree(vertex);
7    for (edge_iterator edge = graphstore.adjacent_edges_begin(vertex);
8         edge != graphstore.adjacent_edges_end(vertex);
9         ++edge) {
10     uint64 target = edge.target_vertex();
11     graphstore.vertex_property(target).work += (pagerank / degree);
12   }
13 }
14 /// --- Update each vertex's PageRank value --- ///
15 for (vertex_iterator vertex = graphstore.vertices_begin();
16      vertex != graphstore.vertices_end();
17      ++vertex) {
18   vertex.property().pagerank = vertex.property().work;
19   vertex.property().work = 0.0;
20 }
```

Figure 4.17: Single Process PageRank Algorithm

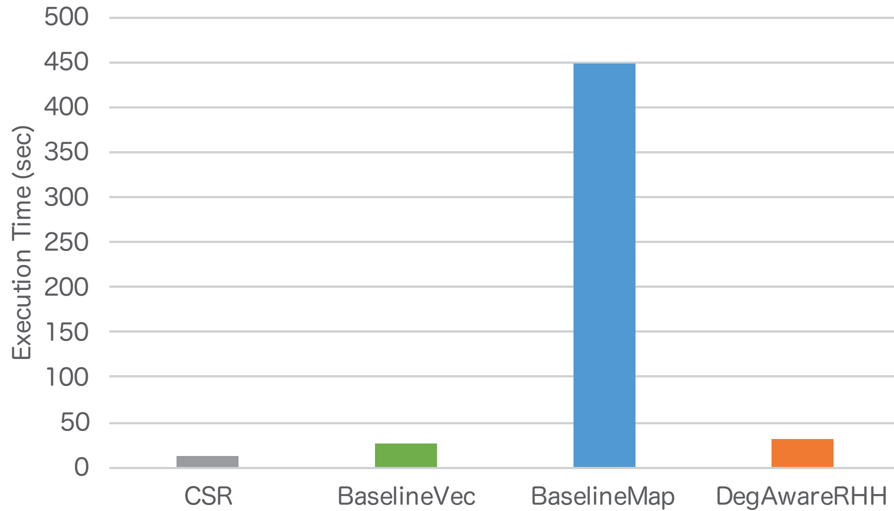


Figure 4.18: The Performance of Single Process BFS

4.7.2.1 BFS

The execution time of the BFS algorithm on the RAMT SCALE 26 graph with the 4 graph data stores are shown in Figure 4.18. As we expected, CSR shows the best performance (11.7 seconds) followed by BaselineVec (26.0 seconds). DegAwareRHH achieves comparable performance (32 seconds) with the 2 implementations, only 2.72x and 1.23x slowdowns against CSR and BaselineVec, respectively. On the other hand, surprisingly, BaselineMap takes much longer time, 449.0 seconds.

4.7.2.2 PageRank

The execution time of the PageRank algorithm on the RAMT SCALE 26 graph with the 4 graph data stores are shown in Figure 4.19. The y-axis denotes the execution time of a super-step on average by running 10 iterations. Again, as we expected, we can see CSR shows the best performance (194.78 seconds). DegAwareRHH takes 653.46 seconds, and it is close to that of BaselineVec, only 8.6% of performance degradation. BaselineMap takes much longer time, 1378.76 seconds.

To conclude this section, we confirmed that DegAwareRHH archives comparable performance on the BFS and PageRank algorithm when compared to the highly packed data structures, i.e., CSR and BaselineVec. We also found that the map (hash) data

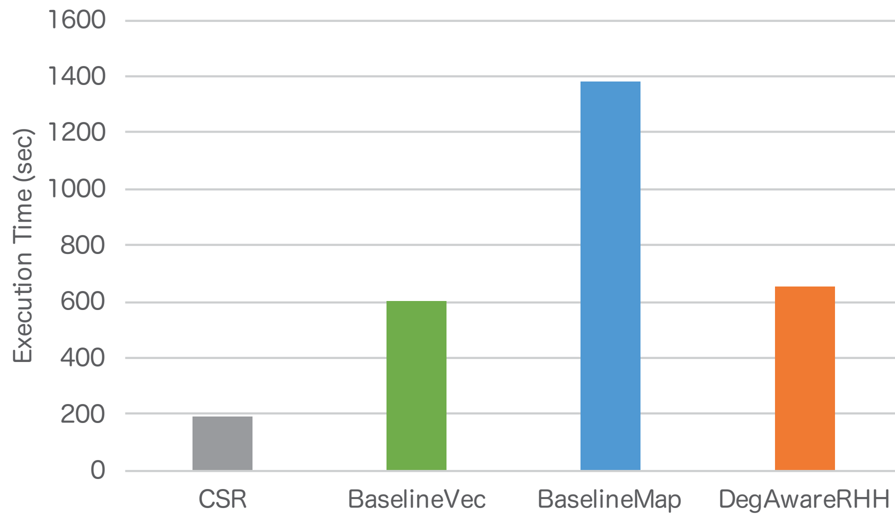


Figure 4.19: The Performance of Single Process PageRank

structure used in BaselineMap, that is unordered_multimap, is not suitable for graph data store due to many pointer accesses when read adjacent edges while DegAwareRHH exhibits satisfactory performance by leveraging the open addressing and linear probing hash table.

4.8 Experiment 3: Dynamic Graph-colouring at Large-scale

In this section, we demonstrate that our DegAwareRHH can accelerate a dynamic graph algorithm at large-scale. We choose graph-colouring as the challenge problem with motivated by its importance and algorithmic properties.

4.8.1 Graph Colouring

Graph-colouring is a problem that colours each vertex in a graph with the minimal number of colours so that no two adjacent vertices have the same colour. Problems in multiple areas that are conceptually an allocation strategy (e.g., scheduling, independence testing, resource allocation) can be represented as graphs: when edges represent scheduling conflicts, a colouring becomes a solution to the allocation problem.

We first introduce the greedy colouring heuristic presented by Jones et al. [48]. Although a heuristic technique doesn't guarantee optimal or perfect result, there are multiple advantages to use heuristic: (1) it is efficient (note that graph colouring is NP-Hard [40]), (2) it allows an asynchronous push-based implementation that fits well on distributed-memory platforms, and (3) it leads to a high-quality solution.

The heuristic works as follows. Each vertex is assigned a priority. There are multiple ways to assign vertex priority, for instance, largest-degree-first (LDF) methods [90], where priorities are determined by vertices' degree. Other strategies, such as a random priority assignment, are also feasible [48].

Each vertex waits for higher priority neighbors to colour themselves and announce their colour, then colours itself based on received neighbors colour information. If two neighboring vertices have the same priority, vertex ID is used to break ties, e.g., the vertex which has lower ID gets a priority. The key advantage this strategy offers is that all uncoloured vertices that are ready to choose their colour (i.e., have maximum priority among their neighbors) can do so concurrently, without any synchronization. This property makes this heuristic ideally suited for a distributed-memory platform.

4.8.2 Distributed Dynamic Graph Colouring Algorithm

Here, we introduce the edge-centoric distributed large-scale dynamic graph-colouring algorithm developed by Sallinen et al. [75].

The algorithm is based on the greedy colouring heuristic presented by Jones et al. [48] (its detail is described in 4.8.1). Each vertex has a priority based on its ID. When an edge is inserted, the vertex who has higher priority colour itself first and announce its colour to the lower priority vertex. At this time, if the higher priority vertex already has its colour, it just announces its colour; if the lower priority vertex already has its colour and the colour conflict with higher priority vertex's colour, choose the next colour which is not used by adjacent vertices.

An example of the dynamic colouring algorithm is described in Figure 4.20, staring the algorithm from a graph which has vertex 1, 2, and 4 as well as edge$(1, 2)$ and edge$(1, 4)$ as its initial state. Vertices chose their colour based on the priority table (illustrated in the right top in Figure 4.20), that is, pick up the highest priority colour which doesn't conflict with any higher priority vertices connecting with. Accordingly, vertex 1 chooses red colour and vertex 2 and 4 choose blue colour.

Step 1: edge$4, 3$ is inserted and vertex 3 chooses its colour as red.

Step 2: edge$1, 3$ is inserted; however, the colours of vertex 1 and 3 are conflicting; thus vertex 3, which has low priority, chooses its colour as orange.

Step 3: edge$3, 4$ is inserted and same as the previous step, based on the vertices' priority, vertex 4 choose its colour as green.

Step 3: edge$1, 3$ is deleted; however, vertex 3 keeps its colour although the number of colours used is not the minimum number.

4.8.2.1 Graph Partitioning

To extend the dynamic colouring algorithm for distributed-memory, the first problem has to solve is graph partitioning problem, that is, determining which process own which vertices. In their paper, the algorithm simply adopts consistent hashing [49] to determine an owner process of a vertex v by computing $v_{id} noduloP$,

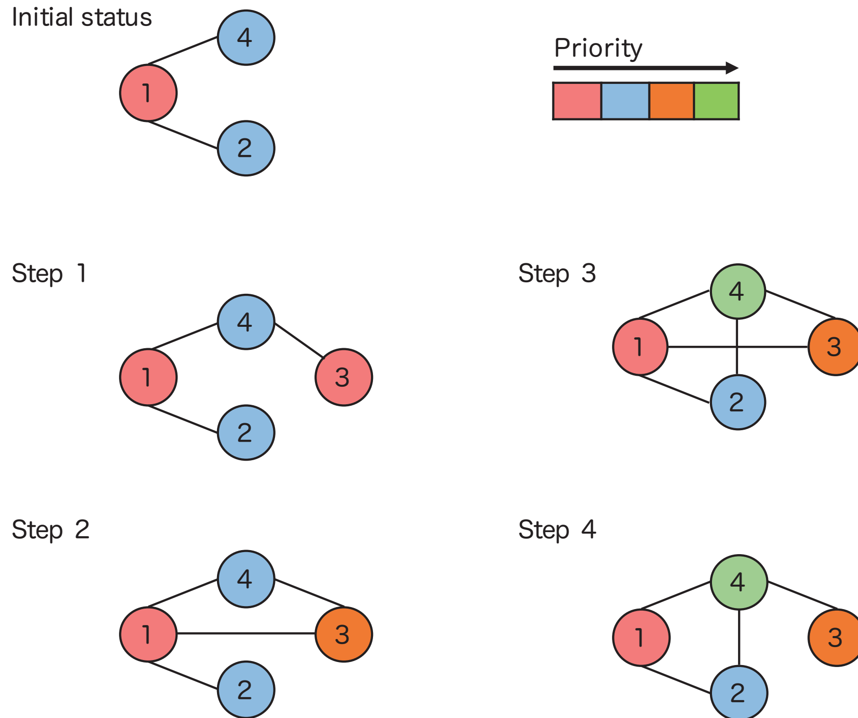


Figure 4.20: An Example of the Dynamic Graph Colouring Algorithm

where P is the number of processes. While consistent hashing is a simple model, it doesn't require any communication between processes and possesses a balanced partitioning in terms of number of vertices. We also use the method for graph partitioning.

4.8.2.2 Experiment Method

In this experiment, we dynamically colour a graph loading and constructing the graph. Thus, graph colouring time includes graph construction time and loading time from disk. We use PowerGraph [41] and BaselineMap for performance comparison.

PowerGraph is a state-of-the-art large-scale distributed-memory graph computation framework written in C++ and developed by the same team created GraphLab [57]. We use their static version of graph colouring implementation. Its execution time is not including graph loading time and construction time. We only time the step for colouring a graph.

4.8.2.3 Machine Configurations

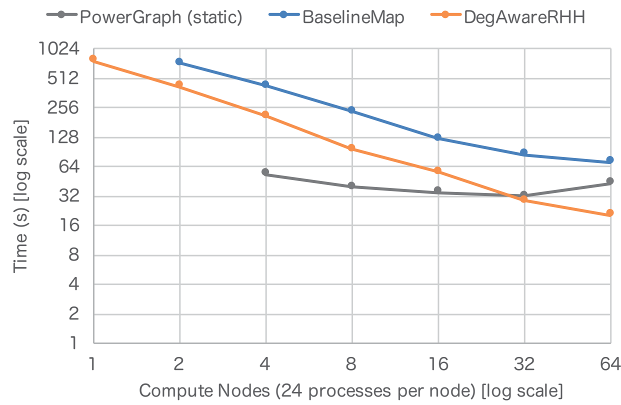
We use Catalyst cluster. All experiments are performed on DRAM (in-core workload only). PowerGraph adopts their own communication implementation that is based on Transmission Control Protocol/Internet Protocol (TCP/IP) instead of MPI; thus, we use IP over InfiniBand (IPoIB) to utilize InfiniBand network of Catalyst cluster, dual rail QDR-80 Intel True Scale Fabric. However, only half lane of the InfiniBand network is used due to the limitation of its implementation.

4.8.2.4 Results

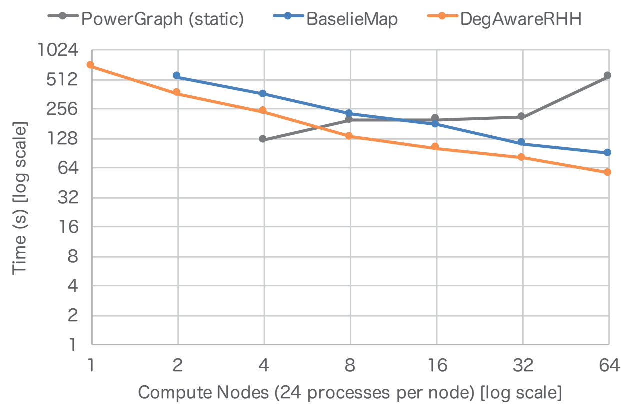
Figure 4.21 shows the result of running the dynamic colouring algorithm on the real-world static graphs, ingesting one parallel stream per process (24 processes per node) same as previous experiments. The y-axis denotes run-time in second (log-scale), and the x-axis denotes the number of compute nodes (24 processes per node). Although direct comparison between PowerGraph and our implementation is not easy, surprisingly, our implementations outperform PowerGraph as the number of compute nodes increases. Also, DegAwareRHH outperforms BaselineMap by 1.33x and 2.03x on the SK500 and Twitter graph at 64 compute nodes, respectively. Notably, using a map (we used a Boost unordered_multimap) is not memory efficient – ending up requiring over 200GB of memory for any of the two graphs; it would not fit into a single node, but does provide a baseline for DegAwareRHH.

The experiment on the Wikipedia dataset with 2 compute nodes and 24 processes per each node is shown in Figure 4.22. Because the Wikipedia dataset is a 'dynamic graph' dataset, we don't use PowerGraph in this experiment. Like the previous experiment, our implementation shows better performance than BaselineMap, 19.14% better performance.

In summary, the results achieved by the experiments indicates DegAwareRHH also exhibits high performance not only dynamic graph construction performance, but also dynamic graph analytics workload.



(a) Twitter



(b) SK2500

Figure 4.21: Run-time for Dynamic Graph Colouring on the Real-world Graphs (24 processes per node, strong scaling)

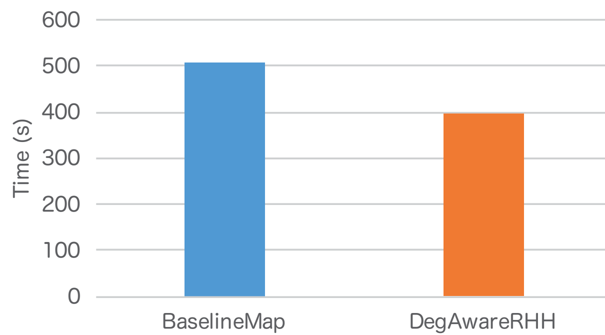


Figure 4.22: Run-time for Dynamic Graph Colouring on the Wikipedia Graph (2 compute nodes, 24 processes per node)

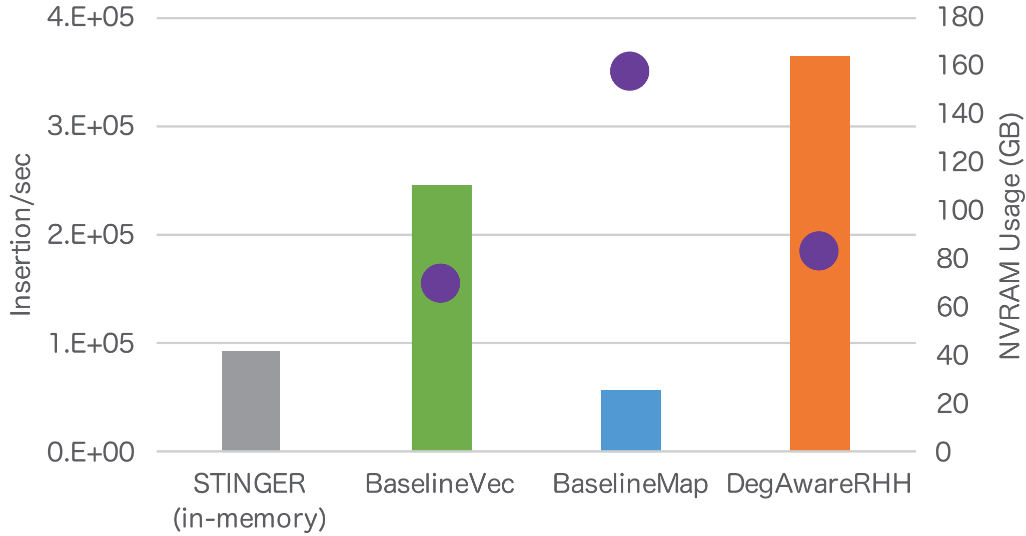


Figure 4.23: Out-of-core Unique Edge Insertion

4.9 Experiment 4: Out-of-core Dynamic Graph Construction

In this section, we experimentally evaluate the performance of out-of-core graph construction workload. We set up the experimental evaluation by restricting the size of available DRAM size by 32 GB in out-of-core workloads by running a dummy program which wastes memory space being aware the NUMA architecture of the machine (i.e., equally wastes memory space among NUMA nodes). We use Roma machine and the installed 4 node-local NVMe 800 GB of NAND flash. For performance comparison, we use BaselineVec and BaselineMap on the out-of-core workload and STINGER with full DRAM capacity (in-core workload). For BaselineVec, BaselineMap, and DegAwareRHH, their entire graph data is stored in NVRAM. We used a RMAT SCALE 26 graph which has 2.3 billion edge insertions and uniquely insert the edges. We show the results in Figure 4.23. The left y-axis and the blue bars denote the number of inserted edges per second; the right y-axis and the violet circles denote the size of used memory size to hold the graph in the NVRAM devices. As can be seen in Figure 4.23, DegAwareRHH outperforms the other implementations: STINGER, BaselineVec, and BaselineMap by 3.8x, 1.5x, and 5.4x, respectively. Regarding the usages of the NVRAM devices, DegAwareRHH used 83.0 GB of memory space in NVRAM which is larger than BaselineVec by approximately 20% and smaller than BaselineMap by approximately half;

DegAwareRHH can construct 2.6x larger graph than available DRAM size with 0.36 million edge insertions per second with 40 processes. To one's surprise, BaselineMap shows worse performance than BaselineVec contrary to the results we achieved in the previous in-core graph construction experiments; this outcome is attributed to not only BaselineMap's larger memory usages but also a lot of pointer accesses in unordered_multimap data structure. On the other hand, owing to high spatial and sequential locality properties and compactness of Robin Hood hashing, DegAwareRHH shows high performance not only in-core but also out-of-core workloads.

4.10 Experiment 5: Dynamic Graph Construction in Future NVRAM Technologies

Finally, we perform an experimental evaluation of dynamic graph construction performance with an emulated future NVRAM device. The emerging NVRAM technologies, referred to as Storage Class memory (SCM), such as PCM and RRAM can be attached to CPU memory bus directly and be expected to possess much lower latency than NAND flash-based NVRAM while keeping lower cost and power consumption and high capacity than DRAM. The emerging NVRAM technologies will have great impacts on out-of-core computing; however, how they affect to the performance of dynamic graph analytics is not clear.

4.10.1 Emulator

To address the issue above, we use a Software Emulator Platform (SEP) provided by Intel which can inject delays into read memory accesses of an application running with the emulator. The architecture of the SEP is illustrated in Figure 4.24. The SEP is a dual socket Ivy Bridge (E5-4620 v2) system with 512 GB of DDR3 1600MHz DRAM, and each CPU socket has 4 memory channels. To emulate future NVRAM devices, the SEP imposes latency to the half number of memory channels of each CPU socket; thus, for each CPU socket, 128 GB of DRAM space is used as

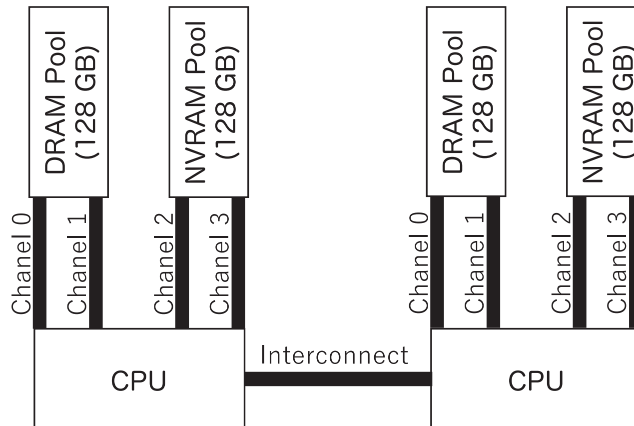


Figure 4.24: Architecture of the Software Emulator Platform (SEP)

emulated NVRAM memory space (NVRAM pool). To vary latencies, the processors are reconfigured with microcode instructions that delay load instructions missing in L3 cache, emulating longer latencies for accessing memory. Although the SEP can emulate only read operations, we argue that our performance observations and analysis still hold true. The processor caches of the SEP system use write back policy, i.e., write I/O is only performed to cache and operation is immediately back to the host. Due to this policy, write induced application latency is minimal for applications exhibiting good locality. The SEP also provides a filesystem like interface while bypassing page cache so that applications using filesystems can be run on the system.

4.10.2 Experimental Setup

We used 8 MPI processes and varied NVRAM read latency from 100ns to 350ns in increments of 50ns. We consider 100ns NVRAM read latency as our baseline instead of DRAM as we observe performance variability with DRAM based experiments due to NUMA-effects. We performed unique edge insertion and deleting workloads on Wikipedia dataset. Same as the previous experiments, the whole structure of the graph store is constructed in the NVRAM pool.

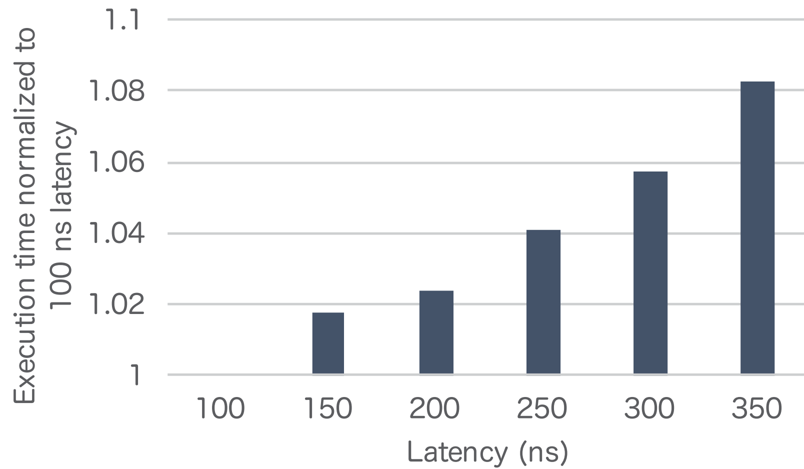


Figure 4.25: Out-of-core Unique Edge Insertion Varying Read Latency From 100 to 350 ns

4.10.3 Result

We show the result in Figure 4.23. The y-axis denotes relative execution time (edge loading time from a local disk is not timed) normalized to the result with 100ns; the x-axis denotes emulated read latency in nanoseconds. As can be seen, as latency increases, the performance of DegAwareRHH increases linearly; however, even at 350ns, performance degradation is only 8% compared with that of 100ns case. The result indicates that our locality-aware design of DegAwareRHH will be efficient in low latency NVM technologies.

4.11 Related work

Many dynamic graph data stores have been proposed based on a tree data structure [81]. However, a tree data structure takes $O(log(n))$ operations and tends to incur more random accesses than Robin Hood hashing, that is a critical matter in large-scale graph processing.

There are many graph databases have been proposed such as *Neo4j* [5], *Giraph* [17], *DEX* [60]. However, several studies have reported that graph databases are slow, have huge memory footprint when considering large-scale graphs, and generally not designed to achieve high performance on graph analytics workloads [33, 61, 42]. A key-value store is a popular database model and is designed to easily scale to a very large size [29]; however, since key-value store doesn't consider the topology of a graph, the locality properties on graph analytics workloads are low.

STINGER [39] is a shared memory (in-core) parallel dynamic graph processing framework and can update a graph with several times–three orders of magnitude better performance in comparison with state-of-the-art 12 open source graph databases and libraries [61], including *SQLite* [66], *Neo4j* [5], *Giraph* [17], *DEX* [60] and *Boost Graph Library* [10]. STINGER stores a graph using adjacency-list model same as DegAwareRHH; however, it stores edges into a linked-block-list (each linked-block has multiple edges) data structure, and it require a sequential search to find a target edge or an empty space for a new edge. Thus, the time to insert an edge (i, j) increases as the out-degree of i increases. In contrast, attributed to adopting a hash table, as we demonstrated in Section 4.6.3, DegAwareRHH outperformed STINGER on a graph construction workload up to 212.2 times.

GraphIn [78] adopts edge-lists to store incremental updates and compressed matrix format for a static version of graph, and it follows a synchronous communication model that is based on the gather-apply-scatter (GAS) programming model [41]. GraphIn firstly run a static graph analytics; secondly marks the portions of the graph that become inconsistent when the update batch is applied; finally runs the incremental graph analytics for the inconsistent area, or re-constructs the static graph with the update batch followed by performing static graph analytics on the updated static graph if the incremental analytic is less efficient. Unlike GraphIn, DegAwareRHH can directly update a graph and run a graph analytics incrementally with high performance without requiring a whole graph re-construction.

Another approach is to accelerate I/O performance itself: maximizing random I/O performance by using multithreading [45], to effectively utilize storage bandwidth by conducting sequence accesses called Parallel Sliding Windows (PSW) method [53], or by using an edge-centric rather than a vertex-centric computation model [74]; combining multiple NVRAM devices to improve bandwidth and IOPS performance without using expensive NVRAM devices [96].

Setting a high threshold for the middle-high-degree vertices will reduce the cost of inserting edges into the middle-high-degree table while causing many hash conflicts in the low-degree table, which are relatively high overhead for low-degree vertices as described in Section 4.3.2.2. Accordingly, from the perspective of improving the performance of Robin Hood hashing itself especially with many hash conflicts, techniques such as using bucket [87] will relieve this problem.

4.12 Discussion: Order of Edge Stream

When evaluating the performance of dynamic graph workloads, both updation and analysis, carefully setting the order of edge streams is important. Specifically, an evaluation with sorted-ordered streams will result in the best performance scenario because the edges from the same vertex continuously access the same edge-list and it contributes to localities. In contrast, random-ordered datasets will result in worse performance. Therefore, we randomized the order of edges in the datasets except the Wikipedia dataset so that to perform fair (or worst case) evaluations; although we would like to address performance evaluations on various real dynamic graph datasets in future work, we expect that our dynamic graph data store will show similar high performance on various dynamic graph workloads.

4.13 Summary and Future Work

4.13.1 Summary

We implemented DegAwareRHH, a high performance dynamic graph data store. DegAwareRHH supports shared and distributed-memory using the asynchronous visitor queue framework. We demonstrated that DegAwareRHH is 212.2 times faster than STINGER, a state-of-the-art shared-memory graph processing framework, with 24 threads/processes on a single node to construct a graph with 1 billion edge insertion requests and 54 million edge deletion requests. We confirmed that DegAwareRHH preserves high graph construction performance on all of our graph construction workloads, unique or non-unique insertions or deletions, even though the size of graphs increase owing to using Robin Hood hashing, an open addressing compact hash tables that exhibit high data locality properties, for its edge-list in contrast to STINGER and Baseline models, which use 1D arrays for their edge-lists. DegAwareRHH also achieved a processing rate of over 1.8 billion edge insertion requests per second at 192 nodes, on a large-scale real graph with 128 billion edges.

For graph analytics workloads, we showed that DegAwareRHH achieves comparable performance even compared with a static compressed data structure, i.e., CSR (Compressed Sparse Row). We also demonstrated that DegAwareRHH can accelerate the performance of the large-scale dynamic graph colouring algorithm and achieve high performance on out-of-core workloads including future NVRAM devices by using the NVRAM emulator.

We designed DegAwareRHH for vertex-centric computation model, wildly used in many graph analytics frameworks and demonstrated high performance on principal graph computation patterns: traversal, graph update and accessing property data. As DegAwareRHH is targeting streaming (incremental) graph analytics so that monitor fine-grained graph changes, we believe that DegAwareRHH will be useful for wide range of incremental dynamic graph analytics.

4.13.2 Future Work

Apply the dynamic graph data store for other dynamic graph algorithms We demonstrated that DegAwareRHH on the asynchronous communication framework has possibility to enable high performance dynamic graph analytics through performing the dynamic graph colouring algorithm as an example of one of the dynamic graph algorithms at large-scale. However, studies for large-scale dynamic graph analytics are still unexplored in terms of how to design graph analytics algorithms on the such infrastructure which support incremental dynamic graph analytics with an asynchronous communication model.

Improve the out-of-core performance of DegAwareRHH Although DegAwareRHH is already designed to support and was able to achieve high performance on out-of-core processing, there is still room for further research into improving its performance.

Dynamic Graph Partitioning on Distributed memory Currently our implementation adopts 1D partitioning, i.e., all edges of a vertex is assigned to the same process where the vertex is located, and the constant hashing to divide a graph into multiple processes. Meanwhile, there are many techniques have been proposed to improve the performance of graph analytics for large scale-free graphs among multi-processes on distributed-memory platforms such as 2D partitioning [26] and dividing high-degree vertices and delegating its part of data and computation to remote nodes [71].

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Recently, the amount of data in the world is growing rapidly. Big data processing brings us many challenges that attributed to not only its volume but also to the emergence of a new paradigm; that is, analyzing the data to discover knowledge, to understand behaviors, and to mine for patterns accompanied with complex memory access patterns on large volume of data. At the same time, demands for large-scale graph analytics has risen as an important kernel for HPC applications in various domains. Due to the explosion of data in the recent years, the size of graphs appear in the real world also has been rapidly increasing. Meanwhile, although NVRAM has negative aspects of low throughput and high latency compare with DRAM, node-local NVRAM has found its way into HPC platforms and have enabled the possibility to extend main-memory capacity without extremely high cost and power consumption to cope with such explosion of data.

However, due to unstructured and fined-grained memory access patterns, large-scale graph analytics suffer from lack of data locality and low memory utilization; a naive implementation of data store for graph analytics will results in significant performance degradation. Therefore, designing locality-aware data stores for large-scale graph analytics is an extremely important key factor to enable high-performance analytics.

To address the above issues, we explored techniques and design of large-scale graph

data stores from the perspective of static and dynamic graph analytics.

First, we explored important techniques for out-of-core static graph stores by developing a high performance out-of-core BFS implementation. We proposed NETALX, an extremely high performance BFS implementation using NVRAM for Hybrid BFS algorithm. NETALX devises the arrangement of graph data on DRAM and NVRAM to improve data locality (reduce the number of accesses to NVRAM) and sequential locality in NVRAM. Specifically, deploys frequently-accessed graph data with fine-grained I/O into DRAM space based on 1) the detailed analysis of memory access patterns of the algorithm; 2) the properties of NAND flash-based NVRAM devices, i.e., fine-grained I/O causes huge overhead. Experimental results of Kronecker graphs compliant to the Graph500 benchmark on a 2-way Intel Xeon E5-2690 machine with 256 GB of DRAM and NAND flash SSDs in RAID 0 showed that NETALX was able to achieve 4.14 Giga TEPS (Traversed Edges Per Second) for a SCALE31 graph problem with 2^{31} vertices and 2^{35} edges, whose size is 4 times larger than the size of graphs that the machine can accommodate only using DRAM, with only 14.99 % performance degradation. We also demonstrate that NETALX can achieve a power efficiency of 11.8 Mega TEPS/W.

Second, for large-scale dynamic graph data store, we proposed DegAwareRHH, a novel high performance large-scale dynamic graph data store, which leverages the linear probing and open addressing compact hash table that exhibits high space and sequential locality in order to minimize 1) the overhead of reading adjacent edges of each vertex; 2) the number of cache misses and page misses, which causes heavy I/O operations to NVRAM while using it. We demonstrated that DegAwareRHH is 212.2 times faster than STINGER, a state-of-the-art shared-memory streaming graph processing framework, on a single compute node to construct a graph with 1 billion edge insertion requests and 54 million edge deletion requests. DegAwareRHH also achieved a processing rate of over 1.8 billion edge insertion requests per second at 192 compute nodes on the massive-scale real graph with 128 billion edges. We also showed that DegAwareRHH can accelerate the performance of the massive-scale dynamic graph colouring algorithm and achieve high performance on out-of-core workloads including future NVRAM devices.

This thesis presents several contributions towards high performance data store for large-scale graph analytics on HPC platforms, including next generation supercomputers which will have locally-attached NVRAM, in terms of locality awareness.

5.2 Future Work

In addition to the future work described in each section, our work also towards for storing more rich graphs (multi-typed property graphs) at large-scale. A property graph is a graph which has property data along with its vertices and edges and may have multiple type of vertices and edges. For instance, when represent Wikipedia's page editing history by graph, there are vertices which correspond to pages, users or categories; edges correspond to activities to pages from users or hyperlinks between pages. Designing high performance large-scale graph data stores for such multi-typed property graphs is challenging.

Bibliography

[1] 3d xpoint technology. https://www.micron.com/about/emerging-technologies/3d-xpoint-technology.

[2] Exascale initiative. http://www.exascaleinitiative.org.

[3] Graph500. http://www.graph500.org/.

[4] Human brain project. http://www.humanbrainproject.eu/.

[5] Neo4j. http://neo4j.com/.

[6] The opte project (the internet 2003). http://www.opte.org/the-internet/.

[7] The seattle internet exchange (six). http://www.seattleix.net/.

[8] Top500. http://www.top500.org. http://www.top500.org.

[9] Worldwidewebsize.com. http://www.worldwidewebsize.com/.

[10] *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[11] White paper: Cisco vni forecast and methodology, 2015-2020. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html, June 2016.

[12] Sophie Achard, Raymond Salvador, Brandon Whitcher, John Suckling, and ED Bullmore. A resilient, low-frequency, small-world human brain functional network with highly connected association cortical hubs. *The Journal of neuroscience*, 26(1):63–72, 2006.

[13] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

[14] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Internet: Diameter of the world-wide web. *Nature*, 401(6749):130–131, 1999.

[15] Luıs A Nunes Amaral, Antonio Scala, Marc Barthelemy, and H Eugene Stanley. Classes of small-world networks. *Proceedings of the national academy of sciences*, 97(21):11149–11152, 2000.

[16] Raja Appuswamy, Matthaios Olma, and Anastasia Ailamaki. Scaling the memory power wall with dram-aware data management. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, DaMoN'15, pages 3:1–3:9, New York, NY, USA, 2015. ACM.

[17] Ching Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 11, 2011.

[18] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[19] Scott Beamer, K Asanovic, and D Patterson. Searching for a Parent Instead of Fighting Over Children: A Fast Breadth-First Search Implementation for Graph500. *Technical Report UCB/EECS-2011-117, EECS Department, University of California, Berkeley*, pages 1–9, 2011.

[20] 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, Salt Lake City, USA, 2012. IEEE Computer Society Press.

[21] Scott Beamer, Krste Asanovic, and David Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization*, IISWC '15, pages 56–65, Washington, DC, USA, 2015. IEEE Computer Society.

[22] Scotto Beamer, Aydin Buluc, Krste Asanovic, and David Patterson. Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search. In *Workshop on Multithreaded Architectures and Applications (MTAAP2013) in conjunction with 27th IEEE International Parallel & Distributed Processing Symposium (IPDPS2013)*, pages 1 – 10, Boston, 2013.

[23] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience*, 34(8):711–726, 2004.

[24] Antal Bosch, Toine Bogers, and Maurice Kunder. Estimating search engine index size variability: A 9-year longitudinal study. *Scientometrics*, 107(2):839–856, May 2016.

[25] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web. *Comput. Netw.*, 33(1-6):309–320, June 2000.

[26] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 65:1–65:12, New York, NY, USA, 2011. ACM.

[27] Pedro Celis. *Robin Hood Hashing*. PhD thesis, Waterloo, Ont., Canada, Canada, 1986.

[28] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *in Proceedings of the Fourth SIAM International Conference on Data Mining. Society for Industrial Mathematics*, pages 442–446, 2004.

[29] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[30] F. Checconi and F. Petrini. Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 425–434, May 2014.

[31] Fabio Checconi, Fabrizio Petrini, Jeremiah Willcock, Andrew Lumsdaine, Anamitra Roy Choudhury, and Yogish Sabharwal. Breaking the speed and scalability barriers for graph exploration on distributed-memory machines. In *Supercomputing*, 2012.

[32] 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.

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

[34] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009.

[35] Phillip EC Compeau, Pavel A Pevzner, and Glenn Tesler. How to apply de bruijn graphs to genome assembly. *Nature biotechnology*, 29(11):987–991, 2011.

[36] Gerald F Davis, Mina Yoo, and Wayne E Baker. The small world of the american corporate elite, 1982-2001. *Strategic organization*, 1(3):301–326, 2003.

[37] Luc Devroye, Pat Morin, and Alfredo Viola. On worst-case robin hood hashing. *SIAM Journal on Computing*, 33(4):923–936, 2004.

[38] Holger Ebel, Lutz-Ingo Mielsch, and Stefan Bornholdt. Scale-free topology of e-mail networks. *arXiv preprint cond-mat/0201476*, 2002.

[39] D. Ediger, R. McColl, J. Riedy, and D.A. Bader. Stinger: High performance data structure for streaming graphs. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–5, Sept 2012.

[40] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[41] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[42] Oshini Goonetilleke, Saket Sathe, Timos Sellis, and Xiuzhen Zhang. Microblogging queries on graph databases: An introspection. In *Proceedings of the GRADES'15*, GRADES'15, pages 5:1–5:6, New York, NY, USA, 2015. ACM.

[43] Mark Granovetter. The strength of weak ties: A network theory revisited. *Sociological theory*, 1(1):201–233, 1983.

[44] Pawan Harish and P. J. Narayanan. *Accelerating Large Graph Algorithms on the GPU Using CUDA*, pages 197–208. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[45] B. Hendrickson and J.W. Berry. Graph analysis with high-performance computing. *Computing in Science Engineering*, 10(2):14–19, March 2008.

[46] Tony Hey, Stewart Tansley, Kristin M Tolle, et al. *The fourth paradigm: data-intensive scientific discovery*, volume 1. Microsoft research Redmond, WA, 2009.

[47] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 78–88, Oct 2011.

[48] Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing*, 14(3):654–669, 1993.

[49] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[50] Naoya Maruyama Katsuki Fujisawa Koji Ueno, Toyotaro Suzumura and Satoshi Matsuoka. Efficient breadth-first search on massively parallel and distributed memory machines. In *2016 IEEE International Conference on Big Data (IEEE BigData 2016)*, 2016.

[51] Pradeep Kumar and H. Howie Huang. G-store: High-performance graph store for trillion-edge processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 71:1–71:12, Piscataway, NJ, USA, 2016. IEEE Press.

[52] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[53] 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.

[54] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *Computer*, 36(12):39–48, Dec 2003.

[55] Jure Leskovec and D Chakrabarti. Kronecker graphs: An approach to modeling networks. *The Journal of Machine Learning Research*, pages 1–58, 2010.

[56] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. *Realistic, Mathematically Tractable Graph Generation and Evolution, Using*

Kronecker Multiplication, pages 133–145. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

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

[58] Grzegorz Malewicz, MH Austern, and AJC Bik. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–145, 2010.

[59] 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.

[60] Norbert Martínez-Bazan, Victor Muntés-Mulero, Sergio Gómez-Villamor, Jordi Nin, Mario-A. Sánchez-Martínez, and Josep-L. Larriba-Pey. Dex: High-performance exploration on large graphs for information retrieval. In *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, CIKM '07, pages 573–582, New York, NY, USA, 2007. ACM.

[61] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A. Bader. A performance evaluation of open source graph databases. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, PPAA '14, pages 11–18, New York, NY, USA, 2014. ACM.

[62] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM.

[63] Stanley Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967.

[64] Michael Mitzenmacher. A new approach to analyzing robin hood hashing. *CoRR*, abs/1401.7616, 2014.

[65] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. Graphbig: Understanding graph computing in the context of industrial solutions. In *Proceedings of the International Conference for High Performance Computing,*

Networking, Storage and Analysis, SC '15, pages 69:1–69:12, New York, NY, USA, 2015. ACM.

[66] Mike Owens and Grant Allen. *SQLite*. Springer, 2010.

[67] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[68] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[69] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[70] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Scaling Techniques for Massive Scale-Free Graphs in Distributed (External) Memory. In *2013 IEEE International Parallel & Distributed Processing Symposium (IPDPS2013)*, pages 825–836, 2013.

[71] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 549–559, Piscataway, NJ, USA, 2014. IEEE Press.

[72] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[73] Stefan Richter, Victor Alvarez, and Jens Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proc. VLDB Endow.*, 9(3):96–107, November 2015.

[74] 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.

[75] Scott Sallinen, Keita Iwabuchi, Suraj Poudel, Maya Gokhale, Matei Ripeanu, and Roger Pearce. Graph coloring as a challenge problem for dynamic graph processing

on distributed systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[76] K. Sato, K. Mohror, A. Moody, T. Gamblin, B.R. de Supinski, N. Maruyama, and S. Matsuoka. A user-level infiniband-based file system and checkpoint strategy for burst buffers. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 21–30, May 2014.

[77] Koji Inoue Takatsugu Ono Hiroshi Sasaki Satoshi Imamura, Yuichiro Yasui and Katsuki Fujisawa. Power-efficient breadth-first search with dram row buffer locality-aware address mapping. In *High Performance Graph Data Management and Processing Workshop (HPGDMP16)*, 2016.

[78] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. Graphin: An online high performance incremental graph processing framework. In *European Conference on Parallel Processing*, pages 319–333. Springer, 2016.

[79] Sangwon Seo, Edward J Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 721–726. IEEE, 2010.

[80] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. Preliminary I/O Performance Evaluation on GPU Accelerator and External Memory. In *GPU Technology Conference (GTC'14) (Poster)*, San Jose, USA, March 2014.

[81] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, pages 114–122, New York, NY, USA, 1981. ACM.

[82] Carlos Diuk Ismail Onur Filiz Smriti Bhagat, Moira Burke and Sergey Edunov. Three and a half degrees of separation. `http://research.fb.com/three-and-a-half-degrees-of-separation/`, February 2016.

[83] Steven H Strogatz. Exploring complex networks. *Nature*, 410(6825):268–276, 2001.

[84] Koji Ueno and Toyotaro Suzumura. Highly scalable graph search for the graph500 benchmark. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 149–160, New York, NY, USA, 2012. ACM.

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

[86] Brian Van Essen, Roger Pearce, Sasha Ames, and Maya Gokhale. On the Role of NVRAM in Data-intensive Architectures: An Evaluation. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 703–714. Ieee, May 2012.

[87] Alfredo Viola. Distributional analysis of robin hood linear probing hashing with buckets. In *International Conference on Analysis of Algorithms DMTCS proc. AD*, volume 297, page 306, 2005.

[88] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 11:1–11:12, New York, NY, USA, 2016. ACM.

[89] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.

[90] Dominic JA Welsh and Martin B Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85–86, 1967.

[91] 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*, page 2. ACM, 2013.

[92] Yuichiro Yasui, Katsuki Fujisawa, and Kazushige Goto. NUMA-optimized Parallel Breadth-first Search on Multicore Single-node System. In *2013 IEEE International Conference on Big Data (IEEE BigData 2013)*, 2013.

[93] Yuichiro Yasui, Katsuki Fujisawa, and Yukinori Sato. Fast and energy-efficient breadth-first search on a single numa system. In JulianMartin Kunkel, Thomas Ludwig, and HansWerner Meuer, editors, *Supercomputing*, volume 8488 of *Lecture Notes in Computer Science*, pages 365–381. Springer International Publishing, 2014.

[94] A. Yoo, A.H. Baker, R. Pearce, and V.E. Henson. A scalable eigensolver for large scale-free graphs using 2D graph partitioning. In *Supercomputing*, pages 1–11, 2011.

[95] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821–829, 2008.

[96] Da Zheng, Randal Burns, and Alexander S. Szalay. Toward millions of file system iops on low-cost, commodity hardware. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 69:1–69:12, New York, NY, USA, 2013. ACM.

[97] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, Santa Clara, CA, February 2015. USENIX Association.