UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
Adapting Data Representations for Optimizing Data-Intensive Applications

Permalink
https://escholarship.org/uc/item/6g5383k5

Author
Kusum, Amlan

Publication Date
2016

Peer reviewed|Thesis/dissertation

eScholarship.org
Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Adapting Data Representations for Optimizing Data-Intensive Applications

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Amlan Kusum

December 2016

Dissertation Committee:

Dr Rajiv Gupta, Chairperson
Dr Iulian Neamtiu
Dr Zhiyun Qian
Dr Zijhia Zhao

Copyright by
Amlan Kusum
2016

The Dissertation of Amlan Kusum is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I wish to express my sincere appreciation to those who have contributed to this thesis and supported me in one way or the other during this amazing journey.

First and foremost I would like to thank my advisors Dr. Rajiv Gupta and Dr. Iulian Neamtiu who were always there for me and shaped my research in many ways. Words cannot sufficiently express my gratitude and thankfulness towards my advisors for their unwavering and consistent confidence in my potential and capabilities. For their understanding and patience during my highs and lows. For their inspiration and guidance in my research, writing and presentation. Without their help, support, guidance and enthusiasm, this would not be a reality.

Next, I would like to thank Dr. Zijhia Zhao and Dr. Zhiyun Qian for being the part of my dissertation committee and reviewing my dissertation. Thank you Dr. Zizhong Chen and Dr. Harsha Madhyastha for being part of my advancement committee and their valuable feedback during the early stages of my research.

I would like to thank Bourns College of Engineering for providing me the opportunity to carry out my doctoral studies. I am forever indebted to National Science Foundation for funding my research work via research grants (CCF-1524852, CCF-1318103, CCF-1149632, and CCF-0963996) to UC Riverside. To all my professors at NIT Rourkela, particularly Dr. Santanu Rath for kindling my interest towards computers and research.

Next, I would like to express my gratitude to all the members of my research group including Vineet, Keval, Sai, Zack, Bo, Farzad, Mehmet, Yan, Kishore, Changui, Tanzir and Yongzian for helping me in many ways. A big *Thank You* to Nikhil, Sharat, Raghu, Ambadi,

Akshay and Pallavi for all these wonderful years. Thank you, Sunil, Samim, Sandeep, Srijit and Atish for everything.

I'm forever indebted and grateful to my parents and in-laws for their love and constant prayers. And, none of this would have been possible without the unconditional care, co-operation and belief from my better-half, *Priyanka*, for understanding and putting up with me through my grad-student life of publications, accepts, rejects, missing out on family and friends.

To my *parents*, for who I am.

To my wife, *Priyanka*, for all the love and support.

ABSTRACT OF THE DISSERTATION

Adapting Data Representations for Optimizing Data-Intensive Applications

by

Amlan Kusum

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2016
Dr Rajiv Gupta, Chairperson

The need for efficiently managing Big Data has grown due the large sizes of data sets encountered by real-world Big Data applications. For example, large-scale graph analytics involves processing of graphs that have billions of nodes or edges. Because of the compute- and data-intensive nature of data analytics, programmers face multiple challenges in trying to achieve efficiency. This dissertation presents two novel approaches for handling data to scale up the performance of Big Data applications.

The first approach supports multiple in-memory physical representations (data structures) for data and dynamically selects, or switches to, the best representation for the given data/workload characteristics. For example, in a graph processing application, where the graph evolves over time, our approach switches to the best data structure as the characteristics of the graph (e.g., graph density) change over time. Similarly, in a key-value store, upon changes in relative frequencies of different types of queries over time, we switch to a more efficient data structure for that query load. Our programming and compiler support produces adaptive applications that automatically switch data structures on-the-fly

with little overhead and without the developer worrying about safety. Our evaluation shows running that our adaptive applications enjoy average speedups of 1.6× for graph applications and average throughput increase of 1.4× for a key-value store, compared to their non-adaptive versions.

The second approach employs data transformations to create alternate data representations to accelerate shared memory and out-of-core applications. A large input graph is transformed into smaller graphs using a sequence of data transformations. Execution time reductions are achieved using a processing model that effectively runs the original iterative algorithm in two phases: first, using the reduced input graph to reduce execution time; and second, using the original input graph along with the results from the first phase for computing precise results. In the context of out-of-core applications this model is used to reduce I/O cost by creating a smaller graph that can be held in memory during the first phase. For parallel graph applications, our approach yields speedups of up to 2.14× and 1.81× for in-memory and out-of-core processing respectively, compared to applications running on untransformed data.

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Dissertation Overview 3
1.1.1 Employing Alternate Data Structures for Holding Data 3
1.1.2 Employing Data Transformations to Create Multiple Data Representations 5
1.2 Dissertation Organization 7

2 Choice of Data Structure Representation **8**
2.1 Choosing Representations for Evolving Graphs 9
2.2 Choosing Representations in a DBMS 13
2.3 Choosing Representations in an Online Multiplayer Game 16
2.4 Choosing Representation in an Object Caching System 19
2.5 Summary 21

3 Employing Alternate Data Structure Representations **22**
3.1 Overview 23
3.2 Static Analysis 30
3.2.1 Safety Analysis 31
3.2.2 Improving Timeliness 33
3.3 Evaluation 35
3.3.1 Effort and Safety of Manual Adaptation 36
3.3.2 Dataset and System Specification 42
3.3.3 Benefits of Adaptation 43
3.3.4 Overhead of our Approach 51
3.3.5 Effect of Late Adaptation 53
3.3.6 Timeliness Improvements due to Static Analysis 54
3.4 Summary 55

4 Employing Data Transformations to Create Multiple Representations 57
4.1 Overview of Our Approach . . . 58
4.1.1 Efficient Input Reduction Transformations . . . 60
4.1.2 Original and Two-Phased Algorithms . . . 61
4.1.3 Example: Single Source Shortest Paths . . . 64
4.2 Input Reduction . . . 65
4.2.1 Transformations for Input Reduction . . . 65
4.2.2 Transformation Properties . . . 68
4.3 Programming for Transformed Graphs . . . 72
4.3.1 Impact of Transformations on Vertex Functions . . . 72
4.3.2 Graph Algorithms . . . 77
4.4 Analysis and Generality . . . 79
4.4.1 Analysis . . . 80
4.4.2 Generality . . . 81
4.5 Summary . . . 83

5 Employing Multiple Data Representations for Multithreaded Graph Processing 84
5.1 Parallel Original and Two-Phased Algorithms . . . 85
5.2 Evaluation . . . 86
5.3 Summary . . . 98

6 Employing Multiple Data Representations for Out-of-core Graph Processing 99
6.1 Out-of-Core Graph Processing . . . 100
6.2 Original and Two-Phased Algorithms . . . 102
6.3 Evaluation . . . 105
6.4 Out-of-Core Graph Partitioning . . . 111
6.5 Summary . . . 112

7 Related Work 114
7.1 Data Structure Selection and Runtime Adaptation. . . . 114
7.2 Data Transformations and Approximate Computing . . . 117
7.3 Out-of-core Graph Processing . . . 120

8 Conclusions and Future Directions 121
8.1 Contributions . . . 121
8.1.1 Alternate Data Structure Representations . . . 121
8.1.2 Data Transformations . . . 122
8.1.3 Out-of-core Computations . . . 123
8.2 Future Work . . . 124

Bibliography 126

List of Figures

2.1 Density Variation of MovieLens evolving graph. 10
2.2 MSSP on Google+ graph with varying density. 13
2.3 DBMS: Execution time with varying the INSERT percentage. 16
2.4 SpaceTyrant with varying Crowding. 18
2.5 Memcached: execution time with varying workload characteristics using JH and CH hashing technique. 20

3.1 High level overview of adapting alternate data structure representation technique. 23
3.2 Excerpt from Minimum Spanning Tree (MST-K). 25
3.4 DBMS: query throughput before, during, and after adaptation. 44
3.5 Graph applications: memory consumption before, during, and after adaptation for BC. 47
3.6 Space Tyrant: throughput of the adaptive version. 50
3.7 Memcached: throughput before, during and after transition. 52
3.8 Normalized execution times of adaptive and non-adaptive versions in the late adaptation scenario. 54

4.1 Graph reduction example for multiple representation of data technique. . . 65
4.2 Two-phased SSSP processing on $\mathcal{G}$ & $\mathcal{G}'$. 66
4.3 Transformations for Input Reduction. 67

5.1 Normalized execution time of two-phased execution for each benchmark-graph-ERP value. 90
5.2 Scalability of Reduction Algorithm w.r.t. ERP and number of threads for TP-50. 91
5.3 Improvement in scalability using the two-phased model with varying number of threads. 92
5.4 Increase in memory footprint. 94
5.5 Relative Error (log scale) vs. Execution Time (sec) for CD. 96
5.6 Actual ERP achieved and speedups achieved while using different transformation sets for 50% ERP. 97

6.1 High level overview of our two-phased out-of-core approach. 101

6.2 Normalized execution time of two-phased execution for each benchmark-graph-ERP value. 108
6.3 Savings in I/O using ERP-70 for PR, GC & CC. 110

List of Tables

2.1 Worst-case complexity of graph representations. 12
2.2 Best graph representation, by density interval. 12
2.3 Runtime performance of representations for DBMS. 15
2.4 Workload characteristic vs. best representation. 16
2.5 Worst-case complexity of sector representations. 17
2.6 Best sector representation, by crowding interval. 19
2.7 Workload characteristic vs. best representation. 21

3.1 Application size and programming effort. 40
3.2 Static analysis results: safe adaptation points discovered (second row) and analysis time (third row). 40
3.3 DBMS: throughput of non-adaptive versions and the adaptive version. . . . 43
3.4 Non-adaptive and adaptive execution times adaptations for MovieLens graph. 46
3.5 Space Tyrant: throughput under input-triggered adaptation. 49
3.6 Memcached: throughput under input-triggered adaptation. 51
3.7 Conversion overhead for alternate data structure representation technique. . 53
3.8 Response time without/with timeliness analysis. 55

4.1 Structural guarantees for each transformation. 70
4.2 Various vertex-centric graph algorithms. 76

5.1 Graph Algorithms. 88
5.2 Input Graphs. 88
5.3 Relative Error for CD. 95

6.1 I/O time analysis for PR and CC graph applications. 101
6.2 Graph Algorithms. 107
6.3 Input Graphs. 107

Chapter 1

Introduction

Modern day computing has been focusing on Big Data due to the massive amount of information that must be processed by a wide range of applications from numerous domains. Current trends show that the amount of information is continuing to grow. For example, Facebook has grown from 50M users in 2009 to 1.55B users in 2015. The developers have continuously focused on increasing scalability of the "Big Data" applications to cater to this need of ever-increasing data size. Due to their data-intensive nature, Big Data applications require large amounts of memory for successful execution. If the memory is insufficient, Big Data applications may crash or end abruptly, thus new approaches are required to meet this challenge.

Even when there is sufficient memory to hold the data, the performance of the application is highly sensitive to the data representations and its characteristics. For example, if programmers of a matrix multiplication application assume abundance of system memory they may choose to use a simple two-dimensional array representation. However, for matrices

that are sparse, a Compressed Column Storage representation [19] is a better choice as it requires less memory and finishes the matrix multiplication operation faster. Hence, there are instances when a fixed (compile time) choice of data structure may not be the most efficient across inputs with varying characteristics.

Moreover, the characteristics of many modern large datasets change over time, e.g., Facebook users (nodes) and friendships (edges) evolve continuously. Other examples include real-world evolving graphs found in the Konect [42] and Snap [31] repositories. A fixed choice of data structure is a ppor match for evolving datasets. Techniques for automatic *data structure selection* have been proposed. Jung et al. developed DDT [29], a dynamic program analysis tool that identifies the data structures used by executing the application binary with the objective of identifying problems in the data structure choice with respect to a particular compiler and microarchitecture. Their follow-up work, Brainy [30], predicts the best data structure for a particular program input and underlying architecture. While the aforementioned techniques predict the best data structure choice, they are ineffective when data characteristics change over the course of execution.

Even when the characteristics of the input do not change during the course of execution, there is still scope for accelerating the execution of data-intensive applications. Many data-intensive computations iterate over the same data, and owing to large data sizes, the resulting data accesses are costly. Example of such applications include graph processing on large social graphs. To address this issue, researchers have proposed techniques to optimize memory usage and increase performance via approximation techniques. Task skipping [65, 64] is one such approximation strategy to reduce the iteration over the large

input graph. However, this *code-centric* approach fails to achieve efficiency since intelligent skipping is difficult as the program *lacks a global view* of input graph characteristics. Therefore, we have developed an approach that employs multiple data representations to optimize data access while producing accurate results.

1.1 Dissertation Overview

This dissertation introduces strategies for addressing the previously mentioned challenges for Big Data applications. First, we propose a runtime approach for selecting and switching between data representations to optimize application performance. Second, we propose a runtime approach for reducing the input size to optimize performance. Finally, we propose extending the above techniques to out-of-core algorithms (that do not run out of memory as they rely on disk storage to hold the full data set and bring parts of it in memory for processing). We now summarize these proposed approaches.

1.1.1 Employing Alternate Data Structures for Holding Data

The performance of data-intensive applications is highly dependent upon the main data structures used. In Chapter 2, we demonstrate, via a wide range of applications and input characteristics, that using a single data structure representation for an entire program run is problematic in many cases: when *input characteristics* vary (e.g., a graph algorithm analyzing an input graph that evolves over time); or when *task characteristics* vary (e.g., workload profile for a key-value store); or when *state characteristics* vary (e.g., an online game with a variable number of players).

For example, on a read-mostly workload, the performance of the Memcached object cache can double by switching to Cuckoo hashing [20, 61], compared to Memcached's default hashing. However, to achieve this performance gain, the entire implementation has to be switched; we allow this switch to be performed on-the-fly. As a second example, consider running six popular graph algorithms (described in Chapter 3) on MovieLens, a naturally-evolving graph containing movie reviews which has 3,979,428 edges and 36,526 vertices in its final state. Alternate data structures can store the graphs—adjacency list, adjacency matrix, or shards—each with its own trade-offs. Using a single data structure representation imposes a typical performance overhead of 22% compared to our adaptive version which uses different representations during different execution intervals. Hence data structure representation cannot be selected *a priori* at compile time—rather, it should be selected at runtime, when the choice of the data structure can be adapted to changing input or task characteristics.

We propose a *programmer-assisted approach* to adaptation in Chapter 3, where data structures and algorithms change on-the-fly, safely and efficiently. Our approach is based on: (1) programming support for transforming off-the-shelf applications into adaptive applications; (2) compile-time analyses that automatically identify program points at which the application can safely switch between alternative algorithms and data structures, relieving developers from a burdensome and error-prone task; and (3) a runtime component that performs on-the-fly switching, allowing the application to select the right implementation to exploit available resources.

Runtime adaptation poses several challenges: guaranteeing the safety of on-the-fly data structure and algorithm changes, imposing low steady-state overhead, reacting quickly to system and input changes, minimizing programmer burden. We address these challenges as follows. Programmers render applications adaptive by indicating the alternate implementations of a certain data structure, the application's main computation loops, and writing conversion functions between the alternate data structures. Programmers, however, do not specify *where* adaptation should be performed, as that could jeopardize safety, substantially increase programmer burden, and reduce opportunities for adaptation. Instead, our infrastructure uses a suite of static analyses to find *safe* adaptation points and increase *timeliness*, i.e., react quickly to a mismatch between the current data structure and the input or workload characteristics.

1.1.2 Employing Data Transformations to Create Multiple Data Representations

We present a general approach for accelerating *parallel vertex-centric iterative graph algorithms* – a class of data- and compute-intensive algorithms that repeatedly process large graphs until convergence. Even though these algorithms are parallel, their execution times can still be large for real-world inputs. Thus there is substantial benefit in approximating them to save processing time. The novel aspect of our two-phased approach is that it is *input data-centric*. In the first phase, the original (*unchanged*) iterative algorithm is applied on a smaller graph which is representative of the original large input graph; this step yields savings in execution time. In the second phase, the results from the smaller graph are transferred to the original larger graph and, by applying the original graph algorithm, *error*

reduction is achieved, possibly converging to the *final accurate results.* The additional time required to process the reduced graph in the first phase pays off, as it is significantly lower than the savings achieved by the second phase.

To reduce the sizes of input graphs, we propose *input reduction* transformations whose application is guided by their impact on *graph connectivity* (i.e., the global structure of the graph). Our analysis of various characteristics of vertex centric algorithms and properties of these input reduction transformations has shown that it is possible to *achieve fully accurate results* for a subclass of graph algorithms, with remaining algorithms produce approximate solutions. In comparison to *algorithmic* works, our approach is more general and in contrast to *code-centric* our approach has two major advantages:

– *Input Data-centric Approximation:* via input graph reduction, we also achieve the effect of skipping computations like the *code-centric* approach. However, since skipping is achieved as a consequence of input graph reduction that is performed as a preprocessing phase, the decision of what to skip is sensitive to the structure of the input graph. In particular, graph *connectivity* guides the application of reduction transformations.

– *Uncompromised Processing Algorithm:* our approach requires *no changes to the core graph analysis algorithm.* The original algorithm is used until convergence on the reduced graph and for a limited number of iterations on the full graph for error reduction. As a consequence, with careful choice of input reduction transformations, the algorithm's capability can remain *uncompromised*, i.e., if the error reduction phase is continued long enough, precise results can be obtained.

Finally, programmers have designed *out-of-core applications* to handle large data (e.g., geographical system information and social networking) which runs into terabytes in size, hence often does not fit in the main memory. These out-of-core applications use persistent storage as an extension of main memory. Specifically, the applications fetch part of input data from disk such that it fits in memory, process, and then store the results back to disk. We propose to extend our two-phased data-centric acceleration technique for out-of-core applications and use parallel vertex-centric graph applications for evaluation. Since most of the execution time in out-of-core execution is spent in fetch and store, one of the key challenge is to reduce the time spent in the I/O. The representative input to be used in the first phase could also possibly not fit in memory, hence out-of-core execution would be necessary for first phase, which might the subsume the savings achieved in two-phase execution.

1.2 Dissertation Organization

The rest of the dissertation is organized as follows: Chapter 2 presents the study pertaining to the impact of data structure choice on applications with respect to changes in input/workload characteristics. In Chapter 3, we present our runtime technique of deploying *Alternate Data Structure* on different classes of the application. Next, Chapter 4 presents our *Transformation* techniques for multiple representation of input graphs. In Chapters 5 and 6 we present the *Two Phase* approach for *multithreaded* and *out-of-core* applications. In Chapter 7, we review related work, and in Chapter 8 we conclude with a summary of our work and future directions.

Chapter 2

Choice of Data Structure Representation

Big Data application performance is greatly impacted by the data structure it uses to hold data in memory. The programmer fixes the data structure during the implementation, considering characteristics of input data and the operations to be performed on the input data. Many existing works aid the programmer in choosing data structures [29, 30, 70, 45, 72]. However, choosing a fixed data structure might lead to poor performance and use of excessive amounts of memory, or even failing to complete, when there is a variation in input or workload characteristics. There are two important aspects that need to be thoroughly investigated. First, how input/workload characteristics vary for different types of applications over the course of execution. Second, whether a fixed data structure is unsuitable for such applications. In this chapter, we demonstrate these two aspects using 4 real-world data-intensive applications: first, a graph application that computes the Multiple Source Shortest

Path (MSSP); second, a database application implementing an Indexed Flat File DB (DBMS); third, an online multi-player game, SpaceTyrant (ST); finally, a popular key-value data store utility, Memcached (MEMC). The goal of our experiments is to quantify the impact of input/workload characteristics on the application for alternate representation of the same data.

2.1 Choosing Representations for Evolving Graphs

Graph processing continues to increase in popularity with the emergence of applications such as social network mining, real-time network traffic monitoring, etc. These applications, due to their data-intensive nature, require large amounts of memory. The large graphs being processed are held in dynamic data structures constructed at runtime. These applications spend a significant portion of their execution time on memory management associated with large data sets. Therefore we observe that the performance and dependability of such applications depends upon how well the choice of runtime data structure matches the input data characteristics, as discussed next.

Let $G = (V, E, W)$ be a graph, where V is the set of nodes, E is the set of edges, and W holds the edge weights. Graph *density*, defined as

$$Density = \frac{2 * |E|}{|V| * (|V| - 1)} \tag{2.1}$$

i.e., the ratio of the number of edges in the graph compared to a fully-connected graph, is a key characteristic; we use percentages to indicate density, where a low percentage indicates a *sparse* graph and a high percentage a *dense* graph. The main data structure (the input graph G) can be represented in different ways. We focus on three representations: Adjacency

List (`ADJLIST`), which stores the outgoing edges of each node in a list; a collection of Shards (`SHARDS`), where each shard contains all edges incident to a distinct subset of nodes in the graph [44, 38]; and Adjacency Matrix (`ADJMAT`), which stores the edge weights in a matrix. MSSP computes the shortest path from each vertex to every other vertex, and it can be implemented using multiple applications of SSSP, with each vertex as source.

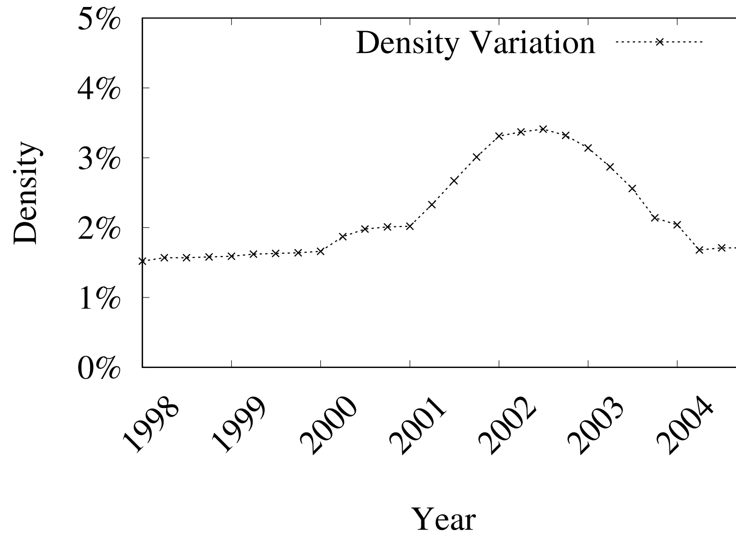


Figure 2.1: Density Variation of MovieLens evolving graph from 1998 to 2004.

Variation in Graph Density. Evolving graphs are a special class of graph data, where edges and vertices are added/deleted over time. Examples of such kind of graphs include social networks and web graphs. In case of social network graphs users join the network (vertices are added) and friendships are created (edges are added). Similarly, new websites (vertices are added) are added to the web with the links are created (edges are added). Along with the size of the evolving graph, other characteristics changes over time as well. We demonstrate this via the real-world evolving graph MovieLens, where the vertices represents

movies and reviewers. The edges represents the reviews from users for a movie. We considered the snapshot of the MovieLens from 1998 to 2004 and plot the density of the graph in Figure 2.1. We can observe that the density of the graph increases from 1.51% at the beginning of 1998 to 3.7% at the end of 2002, and then decreases from 3.7% to 1.71% at the end of 2004. Therefore, we can conclude that for graph input the characteristics could change over period of time.

Space-time Trade-off. Table 2.1 shows the space requirements and edge weight lookup times for the three graph representations. The `ADJLIST` representation exploits graph sparsity to achieve a compact form, but it has the highest worst-case edge weight lookup time, which increases with graph density. The `SHARDS` representation uses additional space and in return provides lower worst-case edge lookup time; note that k_1 depends on shard size and graph density, while k_2 depends on graph density. Finally, the `ADJMAT` representation uses the most space and performs edge weight lookup in constant time, i.e., independent of graph density. Thus, it is expected that input graph density, which is not known at compile time, will affect runtime memory consumption and execution time, and there is an inherent trade-off between space and time among these three representations.

We studied the space and time costs of using the three representations for computing MSSP on a real-world graph, Google+ circles [42]. The graph consists of 23,628 nodes representing users, and 39,242 edges representing links to users in his/her circle. Figure 2.2 plots the execution time and memory consumption for all three representations and demonstrates the expected space–time trade-off among the three representations. For clarity we only show "interesting" graph density ranges, around crossover points (2%, 9%

Table 2.1: Worst-case complexity of graph representations.

Data Structure	**Space**	**Edge Lookup Time**
Adjacency List (`ADJLIST`)	$O(\|V\| + \|E\|)$	$O(\|E\|)$
Shards (`SHARDS`)	$O(\|V\| + k_1\|E\|)$	$O(\|E\|/k_2)$
Adjacency Matrix (`ADJMAT`)	$O(\|V\|^2)$	$O(1)$

Table 2.2: Best graph representation, by density interval.

Criterion	**Density**			
	0-2%	2-25%	25-67%	68-100%
Space	`ADJLIST`	`ADJLIST`	`ADJMAT`	`ADJMAT`
Time	`ADJLIST`	`SHARDS`	`SHARDS`	`ADJMAT`

and 68% for time; 20% and 25% for space), i.e., densities at which one representation starts to outperform another.

Stability across input sizes. To see how the space-time trade-off manifests for representations at different input sizes, we conducted a series of experiments: we varied the graph size from 10,000 to 25,000 nodes, in increments of 5,000; and we varied the density by changing the number of edges. This helps identify crossover points, i.e., threshold densities where one representation starts outperforming another. Table 2.2 summarizes our findings.

The thresholds found in this experiment clarified which representation is a better choice for what density ranges. For example, when the graph density is less than 2%, `ADJLIST` is the "better" representation as it is both more memory- and time-efficient than

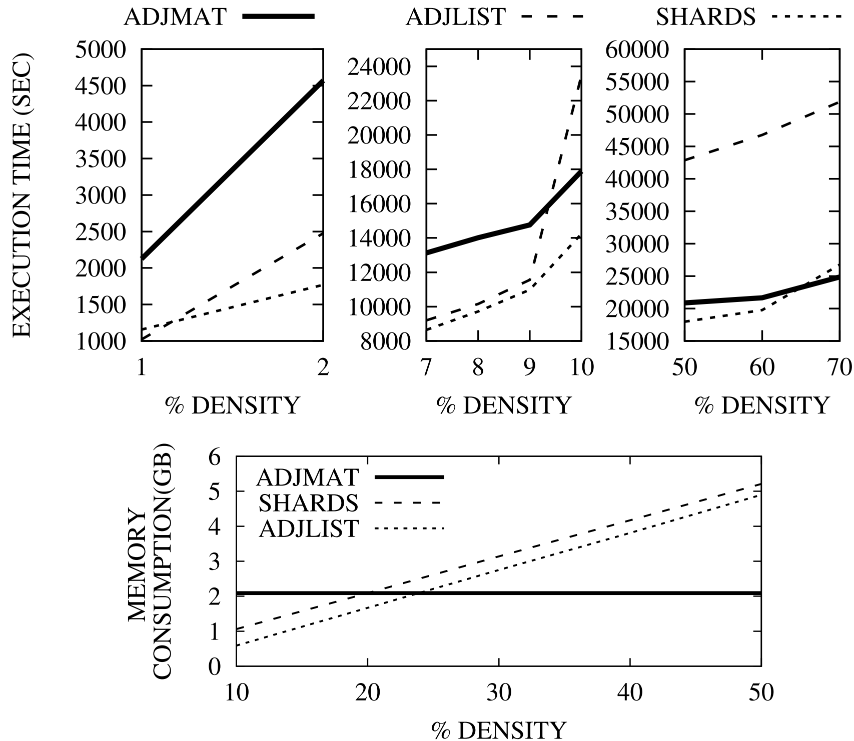


Figure 2.2: MSSP on Google+ graph with varying density: execution time (top) and memory consumption (bottom).

`ADJMAT`. Similarly, in the last interval (> 68%) `ADJMAT` is the clear winner as it takes less memory and is faster. `SHARDS` wins the race of time-efficiency between 2% and 25%. Between 25% and 67%, however, `ADJMAT` is more memory-efficient while `SHARDS` is more time-efficient, hence the best representation depends on characteristics of input graph.

2.2 Choosing Representations in a DBMS

We now illustrate the impact of data structure choice on database operations' performance. For our experiments, the data is stored in a flat file on disk, however the database indexes are stored in memory, in a tree (along with file offsets so data could be

accessed in $O(1)$ time after an offset value is fetched from the tree). There are numerous ways to store the indexes; we chose three popular data structures: BTree (`BTREE`) of order 5, AVL Tree (`AVLTREE`), and Red Black Tree (`RBTREE`). The database operations are real-world social network queries, as explained next.

Variation in Workload. The workload size for a database application may change over a period of time. A typical database would have times when the Database Administrators (DBAs) perform maintenance, which would require more `INSERT` operations than `SEARCH` operations. Similarly, there would be times when the `SEARCH` operations would be far more prevalent than `INSERT` operations. For example, a typical online shopping website would have agents issuing more `INSERT` operations during inventory update, while at other times the number of `SEARCH` operations would vastly outnumber `INSERT` operations. Therefore, a typical workload has a varied mix of `INSERT` and `SEARCH`, which could change over time.

Time Requirements. Although the three indexing data structures have the same worst-case time complexity for insert and search operations—$O(log\ n)$, where n is the number of nodes—they have different insertion and search times due to their rebalancing policy (Table 2.3). For example, `AVLTREE` has a more rigid balance strategy than `RBTREE`, hence taking more time for inserts, but less time for searches. Similarly, in `BTREE`, the searches are faster than `AVLTREE` and `RBTREE` as there is more than one key per node and the nodes are cached from the previous searches, leading to faster searches in the presence of locality. The `BTREE` inserts are also time-consuming as each insert might cause splitting of nodes and/or rebalancing. Table 2.3 summarizes the above observations.

Table 2.3: Runtime performance of representations for DBMS.

Data Structure	**Insert**	**Search**
`BTREE`	Slow	Fast
`AVLTREE`	Slow	Intermediate
`RBTREE`	Fast	Slow

We measured the execution time of insert and search operations on a social network database, with the data and queries from the BG Benchmark [6]. Social networks support a variety of actions, however we consisted of two actions, *initiate friend request* (user X sends a friend request to user Y), and *search friendship* (find if X and Y are friends). *Initiate friend request* requires an insert (SQL `INSERT` operation) while *search friendship* requires a search (SQL `SELECT` operation). We populate the initial network with 10,000 users and 100 friend requests for each user. We generated a workload of 5,000,000 operations which varies the insert/search ratios, from 10% inserts–90% searches to 90% inserts–10% searches. These workloads are run on 3 indexing trees: `RBTREE`, `AVLTREE` and `BTREE`. The goal of our experiments in this section is to show the impact of workload characteristics and indexing data structure on execution time. Similar to the MSSP example, the characterization of program behavior is also simple in this example. In Figure 2.3 we show the execution time for the three index data structures when the workload changes from 10% inserts–90% searches to 90% inserts–10% searches. From the graph we can conclude that for a lower percentage of inserts, `BTREE` performs better than `RBTREE` and `AVLTREE`. Between 37% and 62%, `AVLTREE` is better than the other two. Beyond 62%, `RBTREE` is the best.

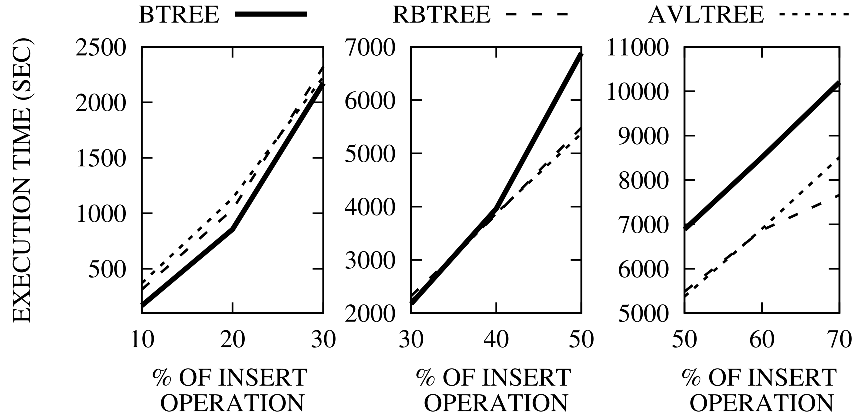


Figure 2.3: DBMS: Execution time with varying the `INSERT` percentage.

Table 2.4: Workload characteristic vs. best representation.

Criterion	**Percentage of Insert Operations**		
	0-37%	37-62%	62-100%
Time	`BTREE`	`AVLTREE`	`RBTREE`

Stability across Workload Sizes. Similar to MSSP, we varied the workload size from 1,000,000 queries to 5,000,000 queries and found that crossover points are stable across workload sizes. We summarize our findings in Table 2.4: each representation has an interval where it is the most suitable.

2.3 Choosing Representations in an Online Multiplayer Game

Space Tyrant is an online multiplayer game server, where players move their ships around a 2D universe. The game state is kept in a *map* divided into *sectors* and information for each sector is stored in a `LIST`, which is backed up periodically on the disk. We found an

alternative, compressed method of representing sectors in memory, `CLIST`, in which only the occupied sectors are stored, along with their neighboring sectors. Let $G = \{U, S\}$ be a game where U is the number of users, and S is the number of points in the space represented as sectors. Game *crowding*, defined as the following,

$$Crowding = \frac{U}{S} \tag{2.2}$$

i.e., the ratio of the number of users playing to the number of sectors is a key characteristic for choosing the map representation.

Table 2.5: Worst-case complexity of sector representations.

Data Structure	**Space**	**Sector Lookup Time**
List (`LIST`)	$O(\|S\|)$	$O(1)$
Compressed List (`CLIST`)	$O(\|U\|)$	$O(\|U\|)$

Variation in *Crowding*. A typical online multiplayer game has users joining and leaving the game in-between a game-play. This would make the game *crowding* change multiple times during game-play.

Space and Throughput Trade-off. Table 2.5 shows the worst-case space complexity and the sector information look-up time for each representation. The `CLIST` representation exploits the crowding property of the map to reduce memory consumption, however it causes sector lookups to cost more. The `LIST` representation requires more space but performs the sector lookup in constant time. Since crowding is unknown during the compile time, the runtime throughput and memory consumption will get affected by the choice of data

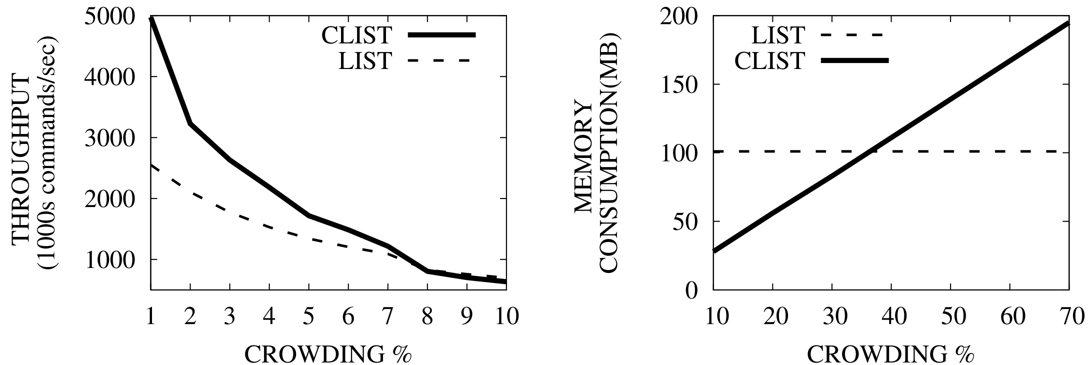


Figure 2.4: SpaceTyrant with varying `Crowding`: execution time (left) and memory consumption (right).

structure, hence the intrinsic time–memory trade-off. We studied the throughput and space costs of the two representations during game plays with 1-million sector maps. For each game play, we varied the number of users from 10,000 to 900,000, thus increasing crowding from 1% to 90%. Figure 2.4 shows that for low *crowding*, i.e., below 8%, `CLIST` is a better representation in terms of both the throughput and the memory consumption; however, above 35%, `LIST` is better in terms of both memory and time.

Stability across Game Sizes. We studied the stability of the crossover points by varying the game size from 500,000 to 1,000,000. The crossover points remain consistent across different game sizes. We summarize the findings in Table 2.6, and we conclude that each representation has an interval where it is the most suitable.

Table 2.6: Best sector representation, by crowding interval.

Criterion	Crowding		
	0-8%	8-35%	35-100%
Space	CLIST	CLIST	LIST
Time	CLIST	LIST	LIST

2.4 Choosing Representation in an Object Caching System

Memcached is a high performance object caching system used by websites such as LiveJournal, Wikipedia, and Flickr. The object caching is achieved by employing a hash table. The hashing technique used in stock Memcached is Jenkin Hash (JH). MemC3 ("Memcached with CLOCK and Concurrent Cuckoo Hashing" [20]) is another variation of Memcached which employs Cuckoo Hashing (CH) [61].

Variation in Workload Characteristics. Atikoglu et al. [3] have analyzed Memcached use at Facebook, one of the largest Memcached deployments. They have found that a typical workload consists of a mix of GET and SET operation. Interestingly, the distribution of GETs:SETs operation varies from 30:1 to 8:37 over time. Therefore, the workload characteristics for Memcached could vary multiple times during execution.

Time Requirements. The hashing technique could greatly impact the performance of Memcached. The authors of MemC3 have measured the relative performance of GET and SET for both hashing techniques (JH & CH), and have concluded JH has faster SETs and slower GETs than CH [20, 61]. A typical workload consists of a mix of GETs and SET commands. We

now study the effect of hashing technique on Memcached performance using different mix of workload, i.e., change the GET:SET ratio. We used YCSB to generate 100 million queries with increasing GET:SET ratio from 10:90 to 90:10. These workload runs on two different hashing techniques and measure the execution time. We report our findings in Figure 2.5, where we plot the execution time of the workload for both hashing techniques.

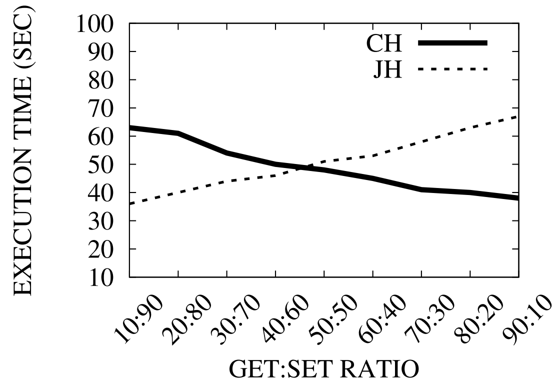


Figure 2.5: Memcached: execution time with varying workload characteristics using JH and CH hashing technique.

We can conclude from Figure 2.5 that for lower percentage of GETs in the workload JH works better than CH. When the GET:SET ratio goes beyond 44:56, the CH hashing technique performs better than JH.

Stability accross Workload Sizes. Similar to previous applications, we wanted to study the stability of crossover point across different workload sizes, therefore we varied the workload size from 10 million queries to 100 million queries with a step of 10 million queries. We found that the crossover points remained consistent across workload sizes. Table 2.7 summarizes our finding: each hashing technique has a distinct range of workload characteristics where it performs the best.

Table 2.7: Workload characteristic vs. best representation.

Criterion	Percentage of GET	
	0-44%	44-100%
Time	JH	CH

2.5 Summary

This chapter presented a detailed study of several applications for alternate representations of the main data structure, under different input/workload characteristics. The study demonstrated that in real world applications a single compile time choice of data structure may not be appropriate because the input/workload characteristics may change over time. We further established the relationship between input/workload characteristics and the optimal data structure choice. This simple characterization of program behavior will guide data structure selection by the runtime system described in Chapter 3. Our proposed system naturally switches from one representation to another to match the changing characteristics of input/workload.

Chapter 3

Employing Alternate Data Structure Representations

When input or workload characteristics change multiple times during execution, a single compile-time data structure choice is limits data processing efficiency. Therefore, switching between alternate data structure representations to continuously match the changing input/workload characteristics is the optimal solution. In this chapter we present a runtime system that automatically switches data structures to match the input/workload characteristics during execution, therefore making the application *adaptive*. However, an *adaptive* application poses several challenges: first, it should not be difficult for the programmer to turn off-the-shelf applications into adaptive ones; second, switching between data structures should not violate type safety or produce incorrect results; third, the switching should be fast enough to respond timely to the input/workload changes.

We show that our proposed technique solves the above challenges. We present the overview of our approach in Section 3.1. We illustrate the effectiveness of our technique via multiple real-world applications and datasets in Section 3.3. We evaluate our technique across multiple dimensions: ease of use, effectiveness, and efficiency.

3.1 Overview

We now present our approach for transforming off-the-shelf applications into adaptive applications that safely and efficiently switch between data structure representations to optimize their operation (higher speed, lower memory usage, etc.) and adapt to changes in input, workload, or state.

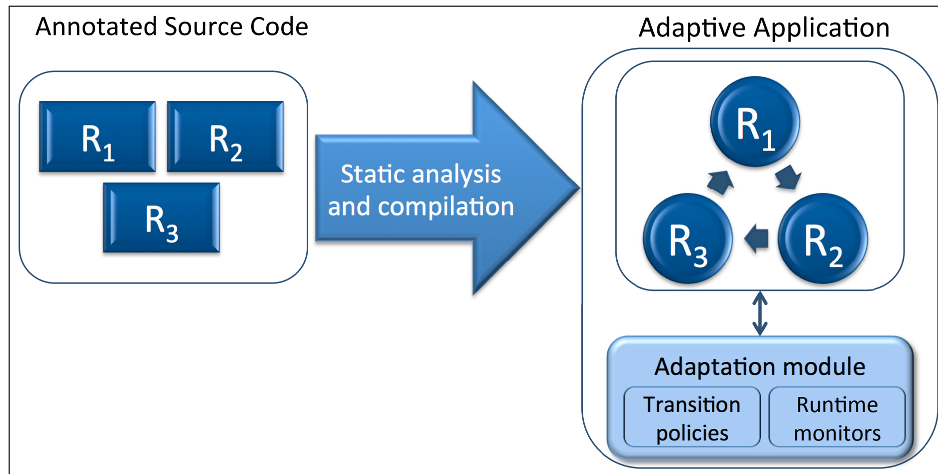


Figure 3.1: High level overview of our approach.

An overview is shown in Figure 3.1; we now describe each step in the process. Programmers need to add a handful of *annotations* to the source code, to indicate alternative representations (data structures & implementations); mark the application's long-running,

compute-intensive loop(s); use a progress indicator (variable holding the value of "progress bar"); and write representation conversion functions. This annotated source code is passed through our *static analyzer and compiler*, which: find safe adaptation points in the source code; perform a source-to-source translation to instrument the code to permit adaptation; and add the transition logic. This code is then compiled with a normal C compiler, e.g., gcc, and then linked with an adaptation module to yield the *adaptive application* that will adapt by safely switching among R_1, R_2, and R_3 (and their associated implementations) at runtime. The *adaptation module* makes adaptation decisions based on system state and/or input characteristics; the conditions under which adaptation should be triggered are defined via transition policies.

Running example: Minimum Spanning Tree using Kruskal's algorithm (MST-K). In Figure 3.2 we show an excerpt from the MST-K application, which computes the minimum weight spanning tree (MST) using Kruskal's algorithm. The input is an undirected graph, while the output is a subgraph whose total weight is less than or equal to every other spanning tree. Three alternate data structure representations are used for holding the large graph in the memory: ADJLIST, ADJMAT and SHARDS, as indicated by the programmer on lines 1–3. The function computeMSTK_ADJMAT, as the name suggests, finds the MST in the ADJMAT representation; similar functions, for ADJLIST and SHARDS representations have been written as well. In each representation, edges have two associated flags. The first flag, "used," indicating whether the edge has been visited to check if it can be added to the spanning tree, is set by markEdgeUsed on line 29. The second flag, "spanningTreeEdge," indicating that the edge is being used in the spanning tree, is set by addEdgeInTree on

```
1   #pragma __ADAPT_DS(GRAPH)("ADJMAT")
2   #pragma __ADAPT_DS(GRAPH)("ADJLIST")
3   #pragma __ADAPT_DS(GRAPH)("SHARD")
4   #pragma __ADAPT_LONG_RUNNING("LR")
5   #pragma __ADAPT_LOOP("AL")
6   typedef struct { //first representation
7   ...
8   } ADJMAT;
9
10  typedef struct { //second representation
11  ...
12  } ADJLIST;
13
14  typedef struct { //third representation
15  ...
16  } SHARD;
17
18  void computeMSTK_ADJMAT( ADJMAT* graph, int* progress)
19  {
20      int totalNodes = graph->totalNodes;
21      Edge* edge;
22
23      LR:{
24          AL: while(*progress < totalNodes)
25          {
26              check4adapt(progress);
27              edge = findMinimumEdge(graph);
28              check4adapt(progress);
29              markEdgeUsed(graph,edge);
30
31              *progress = addEdgeInTree(graph,edge);
32              check4adapt(progress);
33          }
34      }
35  }
36
37  ...
38
39  callOP1(void* graph, int startDS, int* progress) {...}
```

Figure 3.2: Excerpt from Minimum Spanning Tree (MST-K); code in green is inserted automatically by our compiler.

line 30. In addition to nodes and edges, each graph representation also stores the start node of the spanning tree. The progress argument (line 19) holds the value of a "progress bar" for the execution—in this case, the length of the longest tree in the MST (which had no edges initially and the number of nodes is equal to the number of nodes in graph). The algorithm's main part is implemented as a long-running loop LR (line 24) whose counter gets incremented each time an edge is successfully added to the subgraph represented in graph data structure. MST-K is a greedy algorithm, trying to add an edge of minimum value to the tree and terminating when the spanning tree has every node connected (an MST has been constructed); findMinimumEdge (line 27) finds the minimum weight edge in the graph which is not yet in the spanning tree and markEdgeUsed (line 29) marks that edge for possible addition. The function addEdgeInTree (line 31) is two fold: first, it checks if adding the edge to the spanning tree (by setting the spanningTreeEdge true) connects two trees and does not form a cycle; secondly, the spanningTreeEdge is set to true and returns the length of the longest spanning tree. The calls to **check4adapt** (lines 26, 28, and 32) and the definition of **callOP1** (line 37) are inserted automatically by our source-to-source compiler after the static analysis.

Programming Model. Our approach is designed to minimize programmer's burden. To support adaptation, programmers use just four simple annotations, as shown in the following table, to indicate alternative definitions of data structures, as well as long-running code that should be subject to adaptation. Our compiler (static analysis and source code transformation) will use these annotations to generate code that adapts in a safe and flexible manner. In addition, while programmers need to write functions for converting between

representations, they do not need to invoke these functions—for safety and timeliness reasons, these functions are invoked automatically by the runtime system.

Annotation	**Purpose**
progress	progress indicator
__ADAPT_DS(DSname)(ConcreteRep)	mark data type for adaptation
__ADAPT_LONG_RUNNING	long-running block
__ADAPT_LOOP	adaptive loop

In addition to these annotations, the programmer needs to add support for long-running code, as follows. First, the programmer should define a variable (named progress in our examples) which tracks execution progress, e.g., the amount of input processed or the amount of result produced. Second, the long-running computation should be prepared to start processing from a certain progress value rather than assuming it starts from scratch (note the compute MSTK_ADJMAT(ADJMAT* graph, **int*** progress) in Figure 3.2) so the computation can resume at the new representation after a representation switch.

__ADAPT_DS is used to mark a data type for adaptation; the DSname parameter is used as a common name to identify alternate representations, while ConcreteRep indicates the type of the concrete representation. For example, in Figure 3.2 we use the **#pragma** __ADAPT_DS's on lines 1–3 to indicate that ADJMAT, ADJLIST, and SHARD are alternative implementations of the same conceptual type, GRAPH.

The compute-intensive code, usually the program's main loop, is a lexical scope marked via __ADAPT_LONG_RUNNING (lines 4 and 24). This annotation indicates to our

compiler that it should find adaptation opportunities in that scope (or in code transitively called from it, e.g., callees), though the compiler does not descend into loops automatically—this has to be indicated separately, as explained next.

Programmers can annotate a loop with __ADAPT_LOOP to tell the compiler that it should find adaptation opportunities inside that loop's body, i.e., break out of the loop upon the first **check4adapt**(progress),[1] transfer control to the code associated with the new representation, and begin execution from the same "progress" state. In the Figure 3.2 example, if the runtime system indicates a switch is needed from `ADJMAT` to `ADJLIST`, we break out of the loop on line 25 and control is transferred from computeMSTK_ADJMAT to computeMSTK_ADJLIST. Loop annotations are particularly useful when programs contain nested loops: programmers can control the granularity of adaptation, i.e., loops marked with __ADAPT_LOOP will permit fine-grained adaptation.

We have provided the users with minimal number of annotations, so that the burden is relived from the programmer. With fewer changes to the source code, the programmer can rely on our static analysis and compilation tools for the rest of the activities: first, finding out the safe points in the code for switching the data structure; second, making the application react quickly to the changes to input/workload characteristics.

Static Analysis and Compilation. Our static analysis and compiler do the heavy-lifting of carrying out safe and efficient adaptation, which achieves several key goals:

[1](**check4adapt**(progress) is actually an **if** (**check4adapt**(progress))**return**;)

1. Reduce programmer's burden when converting applications into adaptive ones.

2. Automate reasoning about, and enforcing of, adaptation safety.

3. Improve adaptation timeliness.

The results of the static analysis (explained in Section 3.2.1) contain those program points where the transition logic can operate in a manner that there is no type safety violation and the adaptation timeliness is improved.

The source-to-source compiler is responsible for (a) inserting transition code, i.e., the code that will perform the runtime conversion between data structure representations, based on the manual annotations and the results of the static analysis; and (b) inserting potential adaptation points (**check4adapt**). These insertions are guided by pragmas, as explained next. In Figure 3.2, the long-running loop is marked with LR. After static analysis, a **check4adapt**(progress) adaptation check is inserted at appropriate safe points where the adaptation could be triggered. A custom function **callOP1** is added, which is responsible for converting the in-memory data structures to the new representation. All calls to computeMSTK_ADJMAT are then replaced by **callOP1** and the second parameter of **callOP1**, startDS, represents the current data structure representation. When compute-intensive code executes, at **check4adapt**(progress), it checks whether the switch is necessary and if required, the program switches to another representation and the operation is resumed from the point indicated by progress (which in this case is from the length of the longest tree in the minimum spanning tree till last execution).

Adaptation Policies. To specify adaptation points, programmers define a transition policy file that defines which representation should be used for which interval—the system then monitors the input/workload and triggers adaptation automatically. For example, to reduce processing time, the programmer specifies graph density intervals as shown on the left:

```
/* reduce TIME */ or /* reduce MEMORY */
ADJLIST [0,2) ADJLIST [0,25)
SHARD [2,67) ADJMAT [25,100]
ADJMAT [67,100]

/* HYSTERESIS */
TIME 2
```

To reduce memory, the programmer specifies intervals as shown on the right. We also support a hysteresis value (2 seconds in our example); the system waits for the specified time before switching, to avoid too frequent representation changes due to frequent changes in the input characteristics.

Releasing physical memory: When switching representations, after the conversion is finished, our runtime system releases the memory holding the old representation via malloc_trim (similar to application-directed releases [11]). This is particularly important when adapting in response to memory pressure, e.g., from `ADJMAT` to `ADJLIST`.

3.2 Static Analysis

We use static analysis for *safety* (automatically finding points where it is safe to switch representations) as well as *timeliness* (responding faster to adaptation requests). The analysis frees the programmer from worrying about adaptation's safety and timeliness—an intractable manual job for any non-trivial program.

3.2.1 Safety Analysis

The safety analysis prevents the computation code from using mixed data structure representations, as that would be a violation of type safety. We illustrate this on the MST-K example with an excerpt of code from function findMinimumEdge. In a nutshell, the function takes an input graph, sorts its unused edges by calling createSortedEdgeList and returns the first element of the list. We show the function and add a comment to assume we perform a data representation switch from `ADJMAT` to `ADJLIST` at line 51:

```
48  Edge* findMinimumEdge(ADJMAT* graph)
49  {
50  EdgeList* edgeList; // graph in ADJMAT representation
51  // switch ADJMAT to ADJLIST
52  edgeList = createSortedEdgeList(*graph); // type-unsafe!
53
54  return edgeList[0]->edge; ...
```

Clearly, performing a switch at line 51 would violate type safety: since the current function's activation record (findMinimumEdge's stack layout) is set up to assume ∗graph has type `ADJMAT` and createSortedEdgeList takes an `ADJMAT` argument, performing the switch would invoke createSortedEdgeList with an `ADJLIST` argument, which is a violation of type safety—note that `ADJMAT` and `ADJLIST` differ in size and representation hence have different memory layouts.

We solve this problem by enforcing *representation consistency*, a concept originally used to enforce type safety for live program updates [78]. In particular, we use static analysis to annotate each program point with the set Δ of adaptable types used *concretely* in that point's *delimited continuation*[2] and prohibit switching to a new type when the representation

[2]The continuation is delimited by the scope of an adaptive loop, as **check4adapt** can break out of the

assumed by the continuation contains the old type. We now illustrate the analysis by showing the analysis-inferred Δ's in the MST-K example.

```
27  edge = findMinimumEdge(graph);
48  Edge* findMinimumEdge(ADJMAT* graph)
49  {
50      EdgeList* edgeList;
51      Δ = {ADJMAT, ...}; cannot switch
52      edgeList = createSortedEdgeList(*graph);
53      Δ = {...}, ADJMAT ∉ Δ; OK to switch
54      return edgeList[0]->edge;
55  }
```

On line 51, Δ contains `ADJMAT` because the code in the continuation (edgeList = createSortedEdgeList(graph)) assumes the `ADJMAT` representation. The Δ on line 53 does not contain `ADJMAT` as the remaining code in the delimited continuation does not use the graph, hence no representation assumptions are made. To construct Δ's, we have extended the static analysis in [78] to track concrete uses of adaptable data types (as they are marked with an __ADAPT_DS).

Safety condition: We can now provide our formal safety condition: a type-safe switch from representation type τ to τ' can be performed at program point n if:

$$\Delta_n \cap \{\tau\} = \emptyset \tag{3.1}$$

This check is performed statically. In our example, the condition $\Delta_n \cap \{\texttt{ADJMAT}\} = \emptyset$ is satisfied at line 53, hence our compiler will insert a **check4adapt** call to trigger a representation change if needed.

loop, effectively "cutting" the concrete uses in the current iteration or subsequent iterations.

3.2.2 Improving Timeliness

After annotating the program with the type-safety analysis results we are left with a set of program points where a switch is safe. However, a safe switching point does not ensure timeliness, as we will illustrate shortly.

We first introduce some terminology. We name "IR" the intermediate result of the computation, e.g., the partially-constructed spanning tree in the MST-K example. We say that the IR is "dirty" if it has been modified and a representation change will require recomputing the changes made to the IR since the last increment—such recomputations are called "killing" the IR.

The key mechanism we introduce for improving timeliness is to use *contextual effects* [57] to figure out if the IR is dirty and should be killed (in other words, if the last computation increment should be discarded, or can be used before waiting for the next computation increment). In a nutshell, contextual effects are sets that characterize each function and each program point. For functions, the important part of contextual effects is a set named ε that captures whether that function modifies the IR. In MST-K where the IR is stored in graph, some functions, e.g., addEdgeInTree, do write to the IR, hence we have $\{\mathsf{graph}\} \in \varepsilon_{\mathsf{addEdgeInTree}}$; others, e.g., findMinimumEdge, do not write to the IR, hence $\{\mathsf{graph}\} \notin \varepsilon_{\mathsf{findMinimumEdge}}$. The ε effects are chained together to compute, at each program point, a *prior effect* α, i.e., the effect of code that has executed so far, and a *future effect* ω, i.e., the effect of code that will execute. To explain how contextual effects help improve timeliness, we will again use the MST-K example in Figure 3.2. For the sake of this example, let us assume that all points in LR's loop body are type-safe so the representation can be

switched at any point (of course, in practice only type-safe points will be used to improve timeliness). For each line in LR's body, the next code excerpt indicates the prior contextual effect α (which captures whether the program *has modified* graph) and ω (which captures whether the program *will modify* graph):

```
26 α = ∅, ω = {graph}
27 edge = findMinimumEdge(graph); // doesn't modify the IR (graph)
28 α = ∅, ω = {graph}
29 markEdgeUsed(graph,edge);
30 α = {graph}, ω = {graph}
31 *progress = addEdgeInTree(graph, edge);
32 α = {graph}, ω = ∅
```

If we inspect the α and ω annotations on lines 26–32 we see that it is OK to perform the representation switch at lines 26 or 28 without "killing" the graph, because the code has not yet written to graph (findMinimumEdge does not change graph). Similarly, it is OK to perform the switch at line 32 without killing the graph, because the code has written to the graph and will not perform any further writes. However, if we perform the switch at line 30, we have to kill the graph since it is dirty—it has changed and it will change. Hence we can perform a static check to determine whether the switch should kill the IR or not; in the MST-K case, $\{\mathsf{graph}\} \notin \alpha_n \cap \omega_n$. For this, we have extended the static analysis in [57] to track writes to the IR.

Timeliness condition: We can now provide our formal timeliness condition: a type-safe switch can be performed at program point n without killing the IR if:

$$\{IR\} \notin \alpha_n \cap \omega_n \tag{3.2}$$

Safety Proofs and Analysis Infrastructure. Our safety condition is an instance of a property called "con-freeness" while the timeliness condition is an instance of a property

called "transactional version consistency". In this dissertation we just apply these properties — their formal definitions and proofs of correctness can be found elsewhere [78, 57]. The static analyses are inter-procedural, flow-sensitive, though context- and path-insensitive; the pointer analysis is Steensgaard [77]. The analyses and the source-to-source compiler are built on top of the Ginseng infrastructure, which can handle arbitrary C programs [58].

3.3 Evaluation

We evaluate our approach along multiple dimensions. We show that it is *easy to use* (off-the-shelf applications can be converted to adaptive applications with modest programmer burden), *efficient* (applications adapt quickly to changes in input or system characteristics, and their performance is nearly identical to using the best representation at all times) and imposes minimal *time and memory overhead*.

Applications. We used graph algorithms, database operations, and two real-world applications. The six graph algorithms were: Betweenness Centrality (BC) computing the importance of a node in a network; Breadth First Search (BFS), the classical graph traversal; Boruvka's algorithm (MST-B) finds the minimum spanning tree; Preflow Push (PP) finds the maximum flow in a network starting with each individual node as source; MSSP was described in Chapter 2 and MST-K was explained in Section 3.1. The alternative data structure representations were `ADJLIST`, `ADJMAT` and `SHARDS`. For database operations, we used the indexed flat file-based DBMS benchmark described in Chapter 2, with `AVLTREE`, `BTREE` and `RBTREE` as alternative representations. Space Tyrant (ST), an online game server, was described in Chapter 2; the alternate data structures were `LIST` and `CLIST`. Memcached

(MEMC) is a high-performance object caching system used widely in the construction of high-traffic websites. The stock Memcached uses hash tables to store the objects in a key-value store; we name this representation (`JH`) after Jenkin's hash; as an alternate representation we used Cuckoo hashing (`CH`), a complete redesign of Memcached by Fan et al. [20, 61] which can deliver more than double throughput compared to stock Memcached on read-mostly workloads ($\geq 95\%$ reads).

3.3.1 Effort and Safety of Manual Adaptation

Programming effort. Converting an off-the-shelf application into an adaptive application is a four-step process:

Step 1. Identify alternate representations, beyond the existing (single) representation. These alternate representations may already exist in the source code though turned off by a compiler **#define**, or off-the-shelf (e.g., as in Memcached), or have to be implemented.

Step 2. Run the application with a variety of input/workload characteristics to expose the trade-offs and construct the *Adaptation Policy*.

Step 3. Implement alternate representations' conversion functions. If a data structure has N different representations, there will be N*(N-1) conversion functions.

Step 4. Annotate the source code with pragmas, and add support for incremental computation.

We report this effort in Table 3.1. We assume the implementation of alternate representations is available (Step 1) hence we only focus on the programming effort for adaptation itself

(Steps 3 and 4). The conversion code size ("Step 3" column in Table 3.1) depends on the number of alternate representations. For graph applications, the 6 conversion functions for PP amounted to a total of 565 LOC; we were able to reuse that conversion code for the rest of the graph algorithms. The 6 conversion functions for DBMS amounted to 386 LOC, while the 2 conversion functions for ST and MEMC amounted to 215 and 192 LOC, respectively.

"Step 4" code consists of adaptation annotations and support for incrementalization. For annotations (the four grouped columns in Table 3.1) the input data structure names have to be marked using __ADAPT_DS; indicating the IR is not required when it has the same type as the input— this was the case for 8 out of our 9 programs; for BC only, we used one annotation to indicate the IR (column 5). Identifying the long-running section was straightforward for all these applications: we had one such scope per data structure representation (hence 3 per application for graphs and DBMS, 2 per application for ST and MEMC), which we marked with __ADAPT_LONG_RUNNING (column 6). In each long-running scope for graph application, DBMS and ST, we found one loop which needed to be made adaptive, indicated via __ADAPT_LOOP (column 7).

Incrementalization and Other Changes. This effort is shown in the last column of Table 3.1. We modified the compute-intensive functions, so they could be executed in incrementalized fashion (the progress variable from Section 3.1). Increments are "IR units" to be completed toward the final result. Increments have two benefits: (1) enabling the runtime system to stop and start the execution from a particular state and (2) avoiding recomputation after a transition (note that killing the IR means recomputing that increment). An example of incrementalization is shown in Listing 3.3, where we present the original

```
1  void main()
2  {
3

4
5  ...
6
7   computeMSSP(gr);
8  }

9  void computeMSSP(Graph* gr)
10 {
11  int i;
12  for(i = 0;i < gr->numNodes;i++)
13  {
14    sourceID = i;
15    computeSSSP(gr, sourceID);
16  }
17 }
```

(a) Original MSSP.

```
void main()
{
  int* progress =
   (int *) malloc(sizeof(int));
  *progress = 0;
  ...

  computeMSSP(gr, progress);
}

void computeMSSP(Graph* gr, int* prog)
{
  while ( *prog < gr->totalNode )
  {
    sourceID = *progress;
    computeSSSP(gr, sourceID);
    *progress++;
  }
}
```

(b) Incrementalized MSSP.

Listing 3.3: Incrementalization of MSSP code.

Multiple Source Shortest Path (MSSP) implementation (left) and the incrementalized version (right). MSSP is computed as follows: each vertex in the graph is considered as a source vertex and Single Source Shortest Path (SSSP) is computed from that source vertex to every other vertex. In Listing 3.3a, the method computeMSSP computes MSSP as follows: a for loop iterates over all the vertices (lines 12–16) and for each vertex Single Source Shortest Path (SSSP) is computed (line 15). The increments emerge naturally for MSSP, where one unit means computing the SSSP for one vertex. Incrementalization of MSSP (Listing 3.3b) is achieved as follows: first, progress is defined, which stores the vertex id for which SSSP needs to be computed (lines 3–4); second, the progress is added to the computeMSSP method signature so that computation could stop and start from a progress point after adaptation; third, the iterative loop in Listing 3.3b (lines 12–16) is changed to use progress variable for

calculating SSSP from corresponding source vertex stored in `progress`; the `progress` variable is incremented after each unit of MSSP, i.e., after one SSSP is completed. Finally, for Memcached, we had to write 216 LOC to create a new cache manager which can use either hashing technique (`CH` or `JH`); we believe this effort is acceptable, as we effectively had to merge two off-the-shelf Memcached implementations.

Note that although some of our test applications are sizable, only a very small section of code (the main data structure and the compute-intensive functions) needed to be identified and annotated. This was straightforward even though we were not familiar with the code, and we believe it is even easier for developers already familiar with the code.

Analyses' effectiveness. Finding safe and timely adaptation points manually is impractical for any nontrivial program; reasoning about safety is particularly difficult in the presence of nested loops and aliasing. Our two analyses eliminate this programmer burden: in Table 3.2 we show the number of safe adaptation points discovered by our analyses. All these points are type-safe; furthermore, the presence of multiple points increases adaptation timeliness (Section 3.3.6).

Programmer-defined Adaptation Points and their Safety. Our static analyses find program points where representation switching is safe, i.e., type-safe and IR-safe. However, since static analyses must be conservative, we investigated if the programmer could have found better opportunities for adaptation that were missed by our analysis. For this, we manually added adaptation points where we thought it was safe to do so, and then constructed a *dynamic analysis* that traced type and IR accesses to check whether the points

Table 3.1: Application size and programming effort.

Program	**Size** (LOC)	**Step 3** (LOC)	**Step 4**				
			Annotations				Other (LOC)
			DS (input)	DS (IR)	LONG_RUNNING	LOOP	
PP	1,066	565	3	0	3	3	14
MSSP	596	”	3	0	3	3	13
BC	629	”	3	1	3	3	15
MST-K	425	”	3	0	3	3	13
BFS	506	”	3	0	3	3	10
MST-B	795	”	3	0	3	3	19
DBMS	2,566	386	3	0	3	3	6
ST	9,027	215	2	0	2	2	0
MEMC	11,722	192	2	0	2	0	216

Table 3.2: Static analysis results: safe adaptation points discovered (second row) and analysis time (third row).

Program	MSSP	BC	MST-K	BFS	MST-B	PP	DBMS	ST	MEMC
Safe points	3	4	3	3	4	2	4	4	5
Analysis time (sec.)	0.38	0.39	0.24	0.34	0.41	0.36	0.59	8.1	11

were really safe; after the execution, we inspected the trace to find type- and IR-safety violations. We found 1 additional adaptation point in MSSP, BFS and PP; and 2 points in BC that were missed by our static analyses. However, in MST-K (Figure 3.2), line 30, which we thought was safe, was actually found IR-unsafe by the dynamic analysis; of course, line 30 had already been deemed unsafe by the static analyses.

We set out to investigate what would happen if we let programmers pick adaptation points manually: would the points be type-safe and IR-safe? would the adaptation be more timely? We illustrate this with an excerpt of the PreflowPush (PP) application:

```
1   while (*progress < networkGraph->totalNode -1)
2   {
3     resetFlow(&networkGraph);
4
5     startFlow(&networkGraph);
6
7     adjustFlow(&networkGraph, progress);
8   }
```

The PP code computes the flow across each node and edges in a network (graph). The function resetFlow reverts the flow across all the nodes and edges, but assumes that amount of incoming flow to node is equal to outgoing flow. The startFlow function assigns initial excess flows to the all the nodes to start the flow, irrespective of the capacity of outgoing edges. The function adjustFlow adjusts the flow across all the nodes and edges, such that total amount of incoming flow to a node should be equal to the total amount of the outgoing flow. As all three functions modify the IR which is the networkGraph itself, our compiler would insert the **check4adapt** at line 3 and line 9 based on the static analysis. However, if the **check4adapt** is inserted at line 5 manually, and switching is done at that program point, the execution of __ADAPT_LONG_RUNNING loop after the switch will begin with line 3. The

networkGraph which was modified before the switch with the resetFlow, will again try to reset the flow after the representation switch, but it does not modify the networkGraph again (since the flow across all the edges and node is zero). Hence there could be more program points where the data structure switch can occur other the ones found by our tool.

Since the programmer has detailed knowledge of the implementation, she could use her knowledge to insert the adaptation points manually, improving the timeliness of responding to adaptation request, however risking a violation of type-safety, IR-safety, or both. Let us assume the programmer inserts a **check4adapt** at line 7, and a switching is done at that point. After switching the representation, the __ADAPT_LONG_RUNNING loop would resume the execution from resetFlow. This would be incorrect as it violates the *IR*-safety, because the resetFlow expects the networkGraph to have no excess flows. Hence allowing the programmers to define the adaptation points carries a risk of safety violation.

Analysis time. The "Analysis time" row of Table 3.2 presents the sum of static analysis and source-to-source compilation times — at most 11 seconds for our examined applications.

3.3.2 Dataset and System Specification

Real-world Datasets. For graph applications we used real-world graphs from the Konect [42] repository. We used snapshots of MovieLens (evolving graph) from 1999 to 2004; the final snapshot has 3,979,428 edges representing reviews from 8,286 users for 28,240 movies (1.7% density). For the DBMS application, we used the data and queries from the BG Benchmark [6] (Chapter 2). For Memcached we used YCSB [16] to generate key-value queries. For Space Tyrant, we used a large map with various degrees of crowding and a game "controller"

Table 3.3: DBMS: throughput of non-adaptive versions and the adaptive version; values in bold represent the best representation for that workload.

Phase		1	2	3	4	5	Overall
Workload Breakup %INSERT–%SELECT		20–80	50–50	80–20	50–50	20–80	
Non-Adaptive Throughput	BTREE (queries/sec)	**5,847**	725	384	683	**5,151**	860
	RBTREE (queries/sec)	4,752	897	**492**	801	4,231	1,033
	AVLTREE (queries/sec)	4,315	**920**	422	**867**	3,854	980
Adaptive Throughput (queries/sec)		5,725	895	465	843	4,977	**1,035**
Latency (seconds)		1.57	2.12	3.43	5.24		
Overhead (queries/sec)		122	25	27	24	175	

which drives game play by adding/removing users and generating commands for each user.

System Specification. All experiments were run on a 6-core machine (Intel Xeon CPU X5680) with 24GB RAM. This system ran CentOS 5.11 with kernel version 2.6.18-398.el5. Applications were compiled with GCC 4.1.2.

3.3.3 Benefits of Adaptation

DBMS. In this scenario we study how adaptive applications respond to the mismatch between the data structure and workload (query) characteristics. We chose a workload size of 5,000,000 queries partitioned into 5 equal sets (execution phases) with different INSERT–SELECT ratios. The first set has 20% INSERTs–80% SELECTs, while the remaining sets have ratios 50–50, 80–20, 50–50, and 20–80, respectively. We start the program with BTREE; as the workload varies, the adaptive version switches representation as needed.

The results are presented in Table 3.3. Note that the adaptive version's throughput is close to the best non-adaptive version in each phase, as it adapts to the appropriate version

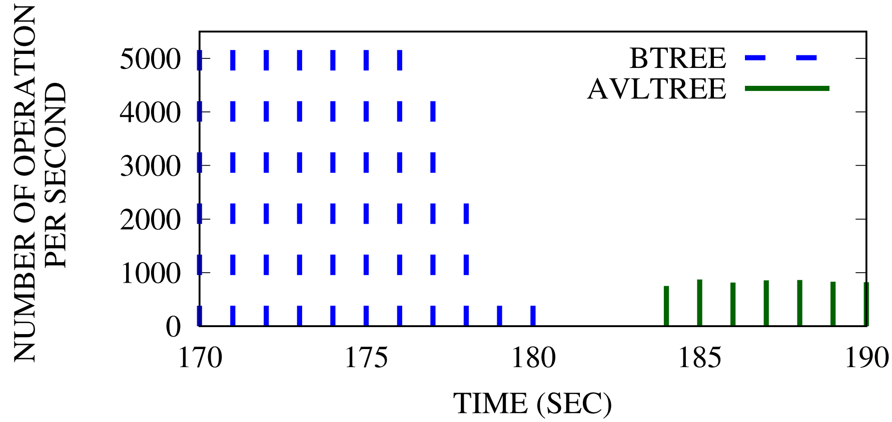


Figure 3.4: DBMS: query throughput before, during, and after adaptation.

shortly after the beginning of the phase. The overall throughput is computed by dividing the total number of queries processed in all 5 phases over total time taken to complete the phases. The overall throughput of the adaptive version is 1,035 queries/sec, virtually the same as the best performing representation (RBTREE at 1,033) and 20% higher than worst-performing representation (BTREE at 860) for the entire execution. For the last phase (phase 5), Table 3.3 contains no latency value as there was no representation change at the end of the execution.

To visualize the adaptation, in Figure 3.4 we show how the throughput varies over time around the interesting (adaptation) region between phase 1 and phase 2, i.e., while changing to a more INSERT-heavy workload. Before the transition, as workload characteristics change, the throughput drops due to the mismatch between the characteristics and the current representation. During the transition, the throughput briefly drops to 0, and then after the transition to AVLTREE, the throughput stabilizes. The figure reveals that (1) the

mismatch is detected early on during the change in workload characteristics, and (2) the transition time is low, relative to the total execution time.

Graph Applications. We use MovieLens as the evolving input graph. The final (year 2004) snapshot has 1.7% density, thus making `ADJLIST` the best representation (lowest memory consumption and execution time). However, at the end of 2002, the density was 3.4%, thus making `SHARDS` the most time-efficient representation. As initial density is less than 2%, the execution starts with `ADJLIST` in the first phase. Then, during execution, the density increases to 3.4% in the second phase, and decreases back to 1.7% in the third phase.

The result of this experiment is summarized in Table 3.4: for each algorithm, we show the maximum density; completion time, in seconds, for the non-adaptive and adaptive versions; the transition latency between phases (again, no transition after phase 2); and the overhead, computed as the difference between phase completion time for the adaptive version and the non-adaptive version at the same representation.

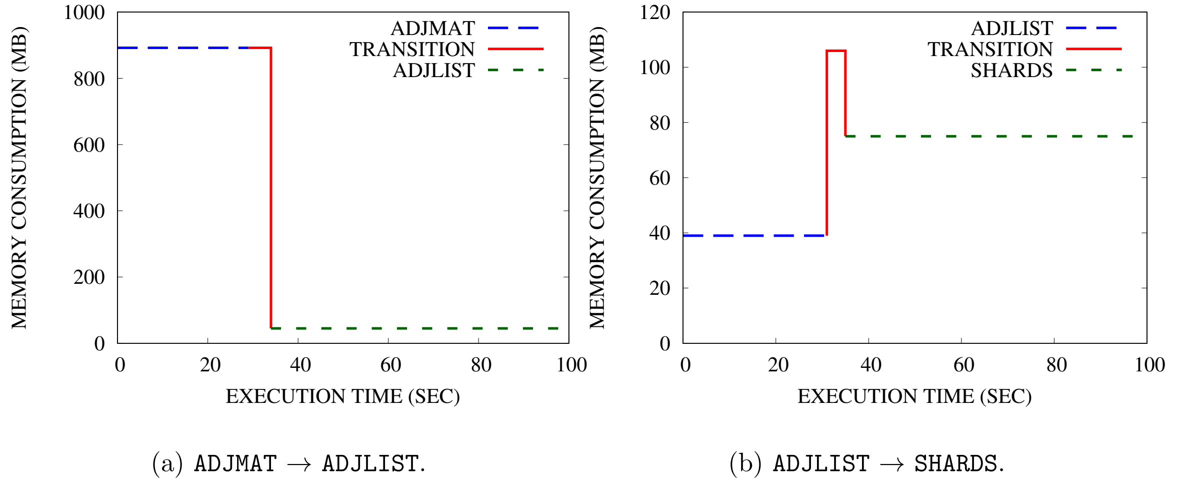
As we can see, during each phase, the performance of the adaptive version is close to the best performing representation in that phase as our system selects the most appropriate version. The overall execution time for non-adaptive versions is calculated as the sum of execution times for each phase. For the adaptive version, the overall execution time is the sum of execution times for each phase, plus the time required for transitions. The overall time of the adaptive version is less than the execution times of the `ADJMAT`, `ADJLIST` and `SHARDS` by an average of 38%, 2%, and 5% respectively. For the PP application, the execution time of the adaptive version is 11% more than `ADJLIST`, and 41% less than `ADJMAT`, the worst choice. Hence, the adaptive version proves to be a better choice than using a

Table 3.4: Non-adaptive and adaptive execution times adaptations for MovieLens graph.

App.	Max Density (%)	Non-adaptive Execution time ADJMAT (sec)	ADJLIST (sec)	SHARDS (sec)	Adaptive (sec)	Latency (sec)	Overhead (sec)
MSSP	2	1,270	**800**	910	837	43	37
	3.4	2,508	1,639	**1,415**	1,445	35	30
	2	2,103	**1,344**	1,654	1,380		36
Overall		5,881	3,783	3,980	**3,668**		103
BC	2	1,197	**691**	803	727	32	36
	3.4	2,399	1,471	**1,278**	1,311	28	33
	2	2,044	**1,204**	1,419	1,241		37
Overall		5,639	3,366	3,500	**3,286**		106
MSTK	2	430	**261**	304	298	14	37
	3.4	844	638	**511**	546	19	35
	2	771	**473**	544	504		31
Overall		2,044	1,372	1,358	**1,354**		104
BFS	2	1,394	**929**	1,041	961	22	32
	3.4	2,718	1,814	**1,695**	1,733	17	38
	2	2,323	**1,678**	1,839	1,718		40
Overall		6,436	4,421	4,575	**4,421**		110
MSTB	2	1,203	**739**	844	778	15	39
	3.4	2,375	1,534	**1,370**	1,404	18	34
	2	2,060	**1,245**	1,548	1,277		32
Overall		5,637	3,518	3,762	**3,466**		105
PP	2	72	**26**	36	35	7	9
	3.4	138	62	**58**	64	1	6
	2	186	**119**	163	125		6
Overall		395	**207**	256	230		21

single data structure for the entire execution. In addition, we observe that the maximum transition latency is just 43 seconds, which represents 1.1% of the execution time for MSSP.

To visualize the adaptation, in Figure 3.5 we show how the memory consumption of graph applications varies over time. We consider two scenarios. First, using the 1999 MovieLens graph, we start in the `ADJMAT` representation. Since its density is low, a mismatch is detected and the representation is switched to `ADJLIST`. Second, we use a MovieLens snapshot from 2002 when it had 3.4% density. Under this scenario we start with `ADJLIST`;



(a) `ADJMAT` → `ADJLIST`.

(b) `ADJLIST` → `SHARDS`.

Figure 3.5: Graph applications: memory consumption before, during, and after adaptation for BC.

our system then switches to `SHARDS`, the most time-efficient representation. In both scenarios, we observe that the mismatch is detected at a very early stage and the system switches to the most efficient representation.

Space Tyrant. We study how the adaptive version quickly rectifies the mismatch between the current crowding level (Chapter 2) and the current data structure. In a real-world gameplay users join, play and leave the game. We used a controller which emulates this scenario and controlled the number of players in the game, thus maintaining the crowding value. We started the game with 1% crowding and `CLIST` representation; after 10 seconds the controller increases the crowding to 10%; after 20 seconds the controller removes players to bring crowding back to 1%. The results of this experiment are summarized in Table 3.5: for each time interval, we show the crowding range; total number of commands executed; the transition latency between intervals (no latency from 1 to 10 seconds as there was no

transition); and the overhead, computed as the difference between the average throughput for the adaptive version and the best non-adaptive version. We can see that, in two out of three time periods, the performance of the adaptive version is better than the non-adaptive version, since there is a mismatch in that time period. In the first period, from 0 to 10 seconds, there was no transition required, since the controller maintained 1% crowding. The low overhead incurred during this time period, (5,000 commands/second, i.e., 0.01%), indicates the efficiency of the adaptive version running with the same representation as the best non-adaptive version. In the second and third time periods, the adaptive version has better performance, since there was a mismatch between the crowding value and its corresponding best representation, hence the overhead is subsumed by increased performance due to switching. The overall throughput is computed by dividing the total number of commands executed in all 3 time periods by 30 seconds. Overall, the throughput of the adaptive version is 3.18% better than `CLIST` and 42% better than `LIST`.

To visualize the adaptation, in Figure 3.6 we show how the throughput varies during game play. Before the first transition, the throughput is decreasing as crowding is increasing from 1% to 10%. When it crosses 8%, the adaptation logic quickly detects the mismatch, and triggers the transition from `CLIST` to `LIST`. Similarly, between seconds 23 and 24, the adaptation logic detects the change in crowding, and triggers the change from `LIST` to `CLIST`.

Memcached. This application is a high-performance object cache used by sites such as YouTube, Facebook, Twitter, and Wikipedia [51]. We considered two different hashing techniques as alternate representations: Jenkin's hash (`JH`) used in stock Memcached, and

Table 3.5: Space Tyrant: throughput under input-triggered adaptation; values in bold represent the best representation for that phase; units for throughput and overhead are thousand commands/sec.

Time		1-10	10-20	20-30	Overall
Crowding		1%	1-10%	10-1%	
Non-Adaptive Throughput (Kcmd/s)	`LIST`	2,551	1,388	1,278	1,739
	`CLIST`	**4,975**	1,958	1,849	2,927
Adaptive Throughput (Kcmd/s)		4,970	**2,241**	**1,860**	**3,024**
Latency (seconds)			0.132	0.206	
Overhead (Kcmd/s)		5			

Cuckoo hashing (`CH`) used in its MemC3 variant [20, 61]. Atikoglu et al. [3] have analyzed Memcached use at Facebook, and found that the distribution of request type ratios (`GET`:`SET`) range from 30:1 to 8:37. `JH` has faster `SET`s and slower `GET`s than `CH` [20, 61]. We studied the behavior of Memcached using both representations on workloads with fixed size (10 million) and found that `JH` is the better representation when the `GET` percentage is below 44% while `CH` is the better representation above 44%.

To realize the benefits of adaptation, we used YCSB [16] to generate 300 million queries, in three phases of 100 million. The `SET`–`GET` split is 70%–30% in the first phase, 30%–70% in the second phase, and 70%–30% in the third phase. Table 3.6 summarizes the results. For each phase, we show the workload characteristics; throughput for non-adaptive and adaptive versions; the transition latency after phases 1 and 2 (no latency for phase 3 since there was no transition). The table shows that during each phase the performance of the non-adaptive version is close to the best-performing representation in the phase, as our system quickly detects the workload characteristics and switches to the best-performing

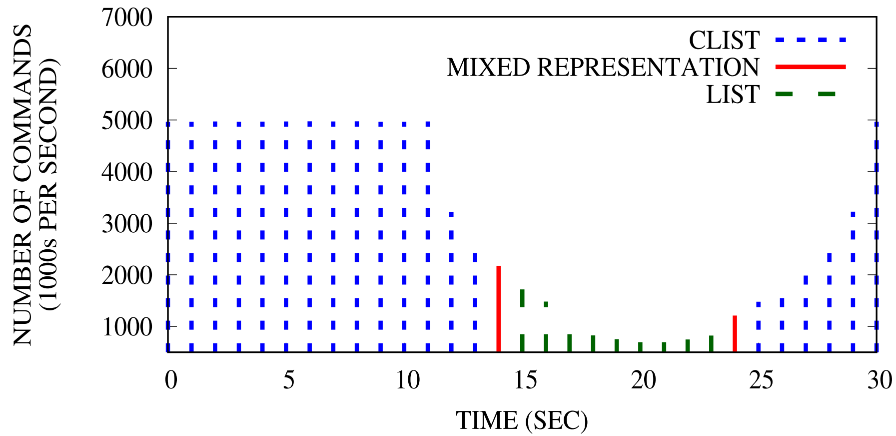


Figure 3.6: Space Tyrant: throughput of the adaptive version.

representation. The overall throughput is computed by dividing the total number of queries processed in all 3 phases over total completion time. The overall throughput of the adaptive version is 2.58% higher than JH and 6.45% higher than CH. The overhead is subsumed by the benefits of adapting to the best hashing technique in each phase; these findings clearly show the benefit of adaptation. Figure 3.7 shows the throughput of the adaptive version: before, during, and after the transition between phase 1 and phase 2. Before the transition the throughput drops due to representation mismatch. During the transition, the throughput drops to around 30,000 queries per second, as the transfer of key-values from one hash to another takes place. After the transition to CH, the throughput increases again. We conclude the following from this figure: first, the mismatch is detected early during the change in workload characteristics; second, although the throughput decreases during the transition, transition time is low compared to the total execution time; third, there is significant performance improvement after the transition.

Table 3.6: Memcached: throughput under input-triggered adaptation; values in bold represent the best representation for that phase; units for throughput and overhead are million queries/sec.

Phase		1	2	3	Overall
Workload Breakup (%GET--%SET)		30–70	70–30	30–70	
Non-Adaptive Throughput (Mqueries/s)	JH	**1.65**	1.27	**1.67**	1.51
	CH	0.95	**1.86**	0.95	1.45
Adaptive Throughput (Mqueries/s)		1.45	1.59	1.64	**1.55**
Latency (seconds)		0.001	0.001		
Overhead (Mqueries/s)		0.25	0.27	0.03	

3.3.4 Overhead of our Approach

The superior performance of the adaptive version is in part achieved because the overhead of our approach is low. The overhead of adaptation has two components: the overhead imposed by the runtime system; and the overhead of switching between the data structure representations.

Runtime Overhead. We measured the overhead of the runtime system as follows. For graph applications, we computed the difference in execution time between the adaptive version and the non-adaptive version for the same data structure in the respective execution phase; the sum of the overheads in each phase gives the execution overhead of the runtime system for a particular graph application. For DBMS, ST, and MEMC, we found the difference in throughput between the adaptive and non-adaptive versions executing with the same representation in the corresponding phase; the average of the overhead in each phase gives the execution overhead. We note from Table 3.3 (last row) that the DBMS

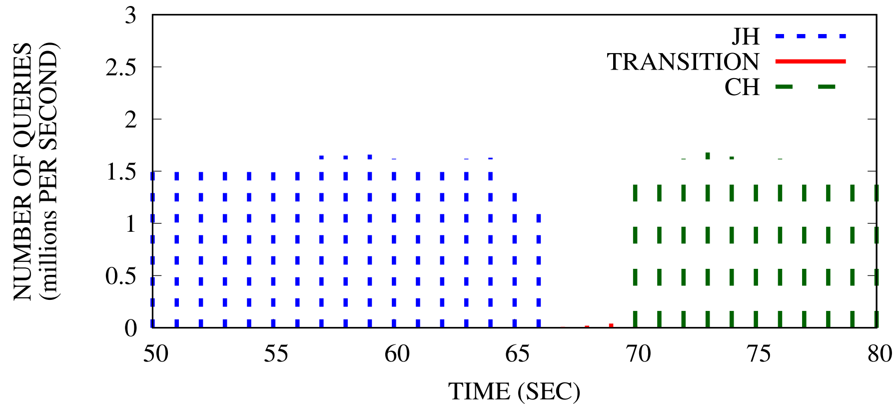


Figure 3.7: Memcached: throughput before, during and after transition.

overhead imposed by our system is on average 4% of the throughput in each phase. Similarly, Table 3.4 (last column) shows that for all graph applications, the execution time overhead imposed by our runtime system is on average 5.2% of the total execution time. Memcached has higher overhead (15%) than other applications, as the mismatch check is performed at the end of every query. The memory overhead of the runtime system itself ranges between 9 to 23 KB, which is negligible compared to the memory used by the rest of the application.

Conversion Overhead. We measured the time and memory overheads incurred during representation changes. For graph applications, we used BC running on MovieLens graph's first snapshot (1.7% density). For DBMS, we used a workload of 5,000,000 queries with 50% `INSERT`s and 50% `SELECT`s with 1,000 initial records in the database. For MEMC we used 100 million queries, 30% `GET`s and 70% `SET`s. For ST, we used the same game play as for measuring the benefits of adaptation. The memory overhead is the additional memory required to carry out the transition between data structures. The time overhead is the duration of the conversion. In Table 3.7 we show the minimum and maximum time/memory

overheads across all adaptations. The results show that conversion time is approximately 1% of the total execution time. We also infer that although there is a memory spike (at most 30.12 MB) for some conversion scenarios, it lasts for a short time (less than 2% of total execution time). The additional memory required to handle transition for Space Tyrant is zero, as there was no additional data structure required to carry out the transition.

Table 3.7: Conversion overhead.

Application	**Conversion time overhead (seconds)**		**Memory overhead (MB)**	
	min	max	min	max
Graph	3.65	5.84	0.35	30.12
DBMS	4.25	6.54	7.62	7.62
Space Tyrant	2.84	2.84	0	0
Memcached	3.16	3.82	0.07	0.07

3.3.5 Effect of Late Adaptation

We now turn to studying the effect of adaptation occurring late in the execution. For this experiment, we used the `wiki-Vote` graph modified to 20% density for which `SHARDS` is the best choice. We consider two scenarios, starting with `ADJMAT` and starting with `ADJLIST`. We turned off the automatic adaptation based on input data monitoring and forced adaptation at the 75% point in the execution. We compare the total execution time in both scenarios with the three non-adaptive versions. In Figure 3.8 we present the execution time, normalized to `ADJLIST`, of `SHARDS`, `ADJMAT`, `ADJLIST` (three leftmost bars) and adaptive versions starting the execution with `ADJMAT` and `ADJLIST` (two rightmost bars). We observe that, for all the applications, even if the adaptation occurs late, the execution

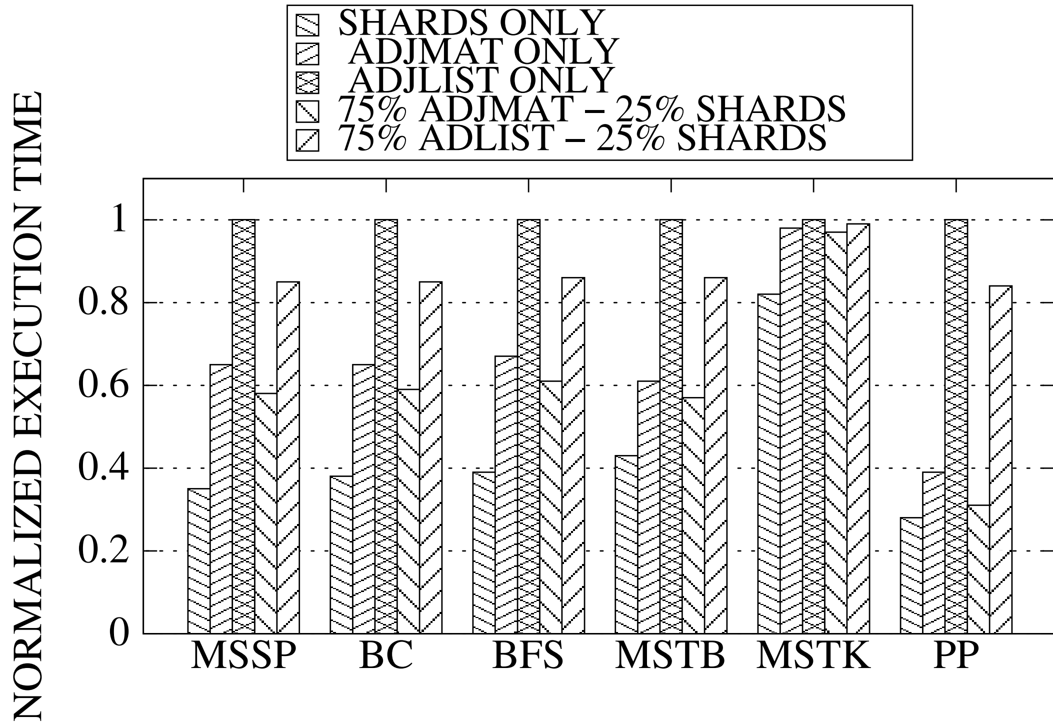


Figure 3.8: Normalized execution times of adaptive and non-adaptive versions in the late adaptation scenario.

time is still in between the worst (`ADJLIST`) and the best (`SHARDS`) representation for the corresponding input graph.

3.3.6 Timeliness Improvements due to Static Analysis

To quantify the benefits of the timeliness-improving static analysis, we ran all the benchmarks as follows: all start in the wrong representation, so an adaptation will be triggered when deemed safe. We measured *response time* as the difference between the time the input monitor has signaled an adaptation and the time when the program reaches a safe adaptation point. We performed this experiment in two settings: first, the program

Table 3.8: Response time without/with timeliness analysis.

Analysis		**Response time (sec.)**						
		MSSP	BC	MST-K	BFS	MST-B	PP	Others
Safety Only	avg.	6.32	2.17	3.62	3.57	4.52	2.18	<0.001
	max	13.87	11.43	8.35	7.42	43.17	4.21	<0.001
Safety & Timeliness	avg.	4.51	1.23	2.17	2.01	1.13	0.57	<0.001
	max	5.11	5.72	3.56	2.77	6.41	1.02	<0.001

with the safety analysis enabled and a single adaptation point in the main loop; second, using our normal compilation scheme, with both the safety and timeliness analyses enabled, hence the compiler could discover additional adaptation points. We present the average and maximum (worst-case) response times in Table 3.8. On average, the timeliness analysis reduces response time by 45%. For Space Tyrant, Memcached and DBMS, (the "Others" column) this benefit is less apparent since adaptation can occur at the end of each query. For graph algorithms, however, where a long-running block consists of several compute-intensive statements, the benefit is clear, e.g., in one MST-B scenario, the response time *without the timeliness analysis was 43.17 seconds* while *with the analysis it was just 0.57 seconds*, out of a total execution time of 874 seconds.

3.4 Summary

In this chapter, we have presented our approach for switching to appropriate data structure representations depending on the input/workload characteristics. In this approach, programmers render applications (constructed from scratch, or off-the-shelf) adaptive by simply indicating the alternate implementations of a certain data structure and the main

computation loops in the application. The programmer, however, does not need to be concerned with specifying where adaptation should be performed, as that could jeopardize safety, substantially increase programmer burden, and reduce opportunities for adaptation. Our infrastructure uses a suite of static analyses to: first, find safe adaptation points in the program; second, increase adaptation opportunities hence increase timeliness, i.e., reduce the time interval between the time where the adaptation need is signaled to when it is effected. The evaluation has demonstrated the effectiveness, ease-of-use and efficiency of our technique, moreover, it has showed that the programming effort required to convert off-the-shelf applications into adaptive applications is modest (9 annotations per application) and the benefits of our technique can be substantial.

Chapter 4

Employing Data Transformations to Create Multiple Representations

Big Data applications process inputs which can easily run into GBs in size even when using the most memory-efficient data structure representation. For example, the social networking graph Friendster (FT) [42] occupies 28GB in memory when stored in the `ADJLIST` representation. Many graph applications are iterative in nature and thus the entire input graph is accessed multiple times making the total cost of data access very high. Completing a widely-used rank computing graph application, Pagerank, on the Friendster graph takes 31 iterations; thus the large input graph is accessed multiple times. Even though graph applications are run in parallel, their execution time can be very high. The performance of such applications can be enhanced by reducing the number of access to the graph.

In this chapter, we propose a novel two-phased processing technique, where we accelerate the execution of *parallel vertex-centic graph applications* by using multiple rep-

resentations of the input graph. The first phase includes processing of a representative input graph which is smaller in size than original input graph. We present our *local* and *non-interfering* reduction technique to extract the representative input from the original input graph. Since the first phase processes the smaller input graph, the number of data accesses is reduced, which in term saves execution time. The second phase uses the original input graph to reduce the error (possibly converging to accurate results), present in results produced by the first phase due to missing vertices and edges.

The rest of the chapter is organized as follows. Section 4.1 presents an overview of our proposed technique. Section 4.2 introduces the technique for input graph reduction – the transformation required to generate the representative input (creating multiple representations of input) and the properties of transformation. In Section 4.3 we present a detailed analysis on how these transformation would affect the vertex-centric graph algorithms to benefit from the two-phased model. Section 4.4 presents: first, a theoretical analysis of the performance benefits that can be achieved by using two-phased approach; second, the generality of two-phased execution technique to accelerate the processing in different scenarios.

4.1 Overview of Our Approach

This section provides an overview of our two-phased processing model. While graph reduction-based processing strategies have been used in various works [26, 52], we focus on iterative general purpose graph algorithms and operate on a single reduced graph along with the original input graph (i.e., there are no multiple levels in the hierarchy). We use

the vertex-centric programming model as it is intuitive and commonly used by many graph processing systems like GraphLab [46], GraphX [88], and Galois [62]. We consider directed graphs in our discussion; our approach can easily be simplified to handle undirected graphs.

Given an iterative vertex-centric graph algorithm $i\mathcal{A}$ and a large input graph $\mathcal{G}$, the accurate results of vertex values $V_{\mathcal{G}}$ can be computed by applying $i\mathcal{A}$ to $\mathcal{G}$, that is:

$$V_{\mathcal{G}} \ = \ i\mathcal{A}(\mathcal{G})$$

To accelerate this computation, we use the following steps:

- Reduce input $\mathcal{G}$ to $\mathcal{G}'$: we transform the large input graph $\mathcal{G}$ into a smaller graph $\mathcal{G}'$ via multiple applications of an input reduction transformation $\mathcal{T}$.
- Compute results for $\mathcal{G}'$: we apply $i\mathcal{A}$ to $\mathcal{G}'$ to compute $V_{\mathcal{G}'}$. Computing on $V_{\mathcal{G}'}$ takes lesser time than on $V_{\mathcal{G}}$.
- Obtain results for $\mathcal{G}$: using simple mapping rules $m\mathcal{R}$s, we convert the results $V_{\mathcal{G}'}$ to $V^1_{\mathcal{G}}$. Then, via multiple application of update rules in $i\mathcal{A}$, we reduce the error in $V^1_{\mathcal{G}}$ and obtain the result $V^2_{\mathcal{G}}$.

Thus, our approach replaces computation $V_{\mathcal{G}} \ = \ i\mathcal{A}(\mathcal{G})$ by:

[INPUT REDUCTION]	$\mathcal{G}' \ = \ \mathcal{T}^{\Delta}(\mathcal{G})$
[PHASE 1]	$V_{\mathcal{G}'} \ = \ i\mathcal{A}\ (\mathcal{G}')$
[MAP RESULTS]	$m\mathcal{R}:\ V_{\mathcal{G}'} \rightarrow V^1_{\mathcal{G}}$
[PHASE 2]	$V^2_{\mathcal{G}} \ = \ i\mathcal{A}\ (V^1_{\mathcal{G}},\ \mathcal{G})$

where Δ is a parameter that controls the degree of reduction performed as it represents the number of applications of $\mathcal{T}$ to $\mathcal{G}$. Thus, the greater the value of Δ, the smaller the size of

the reduced graph $\mathcal{G}'$. Depending on various properties of input reduction transformations $\mathcal{T}$ (Section 4.2.2) and the nature of iterative algorithm $i\mathcal{A}$, the computed values will be accurate, i.e., $V^2_{\mathcal{G}} = V_{\mathcal{G}}$. However, we identify cases in which $V^2_{\mathcal{G}}$ may not be the same as $V_{\mathcal{G}}$ (Section 4.3) — the computed results are *approximate* for those cases.

4.1.1 Efficient Input Reduction Transformations

Given the iterative nature of algorithms considered, applying $i\mathcal{A}$ to $\mathcal{G}'$ as opposed to $\mathcal{G}$ is expected to result in execution time savings. However, these savings can be offset by the extra overhead due to application of input reduction transformations and result converting rules. Therefore we must ensure that these steps are simpler than the iterative computation that they aim to avoid. We do so by placing the following restrictions on the kind of transformation that is allowed (*local*) and the sequence of its application (*non-interfering*) permitted for reducing $\mathcal{G}$ to $\mathcal{G}'$.

A: Local transformation. Transformation $\mathcal{T}(v, \mathcal{G})$, where v is a vertex in $\mathcal{G}$, is a *local* transformation if its application only examines edges directly connected to v. The subgraph involving v and its edges is denoted as subGraph($\mathcal{T}(v, \mathcal{G})$).

$$\mathcal{G}_1 \leftarrow \mathcal{T}(v_1, \mathcal{G}); \quad \mathcal{G}_2 \leftarrow \mathcal{T}(v_2, \mathcal{G}_1) \quad \cdots$$

$$\cdots \quad \mathcal{G}_{\Delta-1} \leftarrow \mathcal{T}(v_{\Delta-1}, \mathcal{G}_{\Delta-2}); \quad \mathcal{G}' \leftarrow \mathcal{T}(v_{\Delta}, \mathcal{G}_{\Delta-1})$$

B: Non-interfering sequence. $\mathcal{T}^{\Delta}$, a sequence of Δ applications of local transformation $\mathcal{T}$ as shown above is *non-interfering* if and only if: vertices $v_1 \cdots v_{\Delta}$ are distinct vertices in $\mathcal{G}$; and each subGraph($\mathcal{T}(v_i, \mathcal{G})$) is contained in $\mathcal{G}$. Note that the above restrictions (local

and non-interfering) ensure that input reduction is performed via a single pass over the original graph because:

- An edge $v_i \rightarrow v_j$ from $\mathcal{G}$ is only examined when considering the application of $\mathcal{T}$ to v_i or v_j; and
- Any vertex or edge created during one application of $\mathcal{T}$ cannot be involved in any other application of $\mathcal{T}$.

Thus, the cost of applying the transformation sequence is linear in the size of $\mathcal{G}$, i.e., the number of vertices and edges in it. Moreover, the cost of converting results is proportional to the size of the transformed portions of $\mathcal{G}$. In contrast, those computations over the transformed portions of $\mathcal{G}$ that we avoid would have required repeated passes due to the iterative nature of graph algorithms considered.

In conclusion, the restrictions on transformations and sequences ensure that the cost of applying them will be less than the cost of the computation they avoid, leading to net savings in execution time.

4.1.2 Original and Two-Phased Algorithms

Next we summarize our approach by presenting the general form of an original iterative vertex-centric graph algorithm (Algorithm 1) and its corresponding two phased version (Algorithm 2). In Algorithm 1, function `IA` represents the original algorithm whose application to graph $\mathcal{G}$ produces the accurate ($V_{\mathcal{G}}$) result. In Algorithm 2, only the functions specific to two-phased approach is shown, while other function which is present original iterative vertex is omitted. Function `TPIA` in Algorithm 2 is the two phased version that

calls `iA` (Algorithm 1) and `iAP2` in first and second phases. Note that the processing logic in `iAP2` (lines 32-37) is exactly the same as that in `iA` (lines 3-6). The result ($V_{\mathcal{G}}^2$) is obtained from the application of `TPiA` to $\mathcal{G}$. The result obtained from `TPiA` might not be accurate; we discuss this in Sections 4.3.1 and 4.3.2.

Algorithm 1: Iterative Vertex-Centric Graph Algorithm.

```
1  Function iA (input G)
2      Initialize V_G & WorkQ
3      while ( ! WorkQ.empty ) do
4          v ← WorkQ.getFirst()
5          if ( UpdateVals(v, V_G)) then
6              WorkQ.add ( outNeighbors (v) )
7      return V_G
8  Function UpdateVals(v, V_G)
9      Updated ← false
10     if ( updateCheck(v, inNeighbors(v) ) then
11         update V_G[v]
12         Updated ← true
13     return Updated
```

ReduceGraph examines the vertices in $\mathcal{G}$ one at a time and if $\mathcal{T}(v, \mathcal{G}')$ is non-interfering with transformations already applied, then it is applied on v. The function `NI` enforces non-interference by ensuring that all vertices and edges in subGraph($\mathcal{T}(v, \mathcal{G})$) are being examined for the first time. The algorithm terminates after applying Δ transformations. The function `iAP2` copies results from vertices in $\mathcal{G}'$ to vertices in $\mathcal{G}$ for each vertex that is present in both graphs. The vertices in $\mathcal{G}$ that were eliminated in the process of creating $\mathcal{G}'$ are assigned initial values by `initval()`. Then, similar to `iA`, UpdateVals is applied to $V_{\mathcal{G}}^2$ until convergence.

Algorithm 2: Two-Phase Iterative Vertex-Centric Graph Algorithm.

```
1  Function TPiA (input 𝒢)
2  |  𝒢' ← ReduceGraph ( 𝒢, 𝒯, Δ )
3  |  V¹_𝒢 ← iA(𝒢')
4  |  V²_𝒢 ← iAP2(V¹_𝒢, 𝒢)
5  |  return V²_𝒢
6  end
7  Function ReduceGraph (𝒢, 𝒯, Δ)
8  |  𝒢' ← 𝒢
9  |  for ( Vertex v : 𝒢 ) do
10 |  |  if ( NI ( subGraph(𝒯(v,𝒢)) ) then
11 |  |  |  𝒢' ← 𝒯(v,𝒢')
12 |  |  |  Δ ← Δ − 1
13 |  |  |  if ( Δ == 0 ) then
14 |  |  |  |  break
15 |  |  |  end
16 |  |  end
17 |  end
18 |  return 𝒢'
19 end
20 Function iAP2(V¹_𝒢, 𝒢)
21 |  Initialize WorkQ
22 |  for ( Vertex v : 𝒢 ) do
23 |  |  if ( v ∈ 𝒢' ) then
24 |  |  |  V²_𝒢 ( v ) ← V¹_𝒢 (v)
25 |  |  end
26 |  |  else
27 |  |  |  V²_𝒢 ( v ) ← initval ( )
28 |  |  |  WorkQ.add ( v )
29 |  |  end
30 |  end
31 |  WorkQ.add ( Vertex v s.t. v is affected by addition / deletion of edges)
32 |  while ( ! WorkQ.empty ) do
33 |  |  v ← WorkQ.getFirst()
34 |  |  if ( UpdateVals (v, V_𝒢) ) then
35 |  |  |  WorkQ.add ( outNeighbors (v) )
36 |  |  end
37 |  end
38 |  return V_𝒢
39 end
```

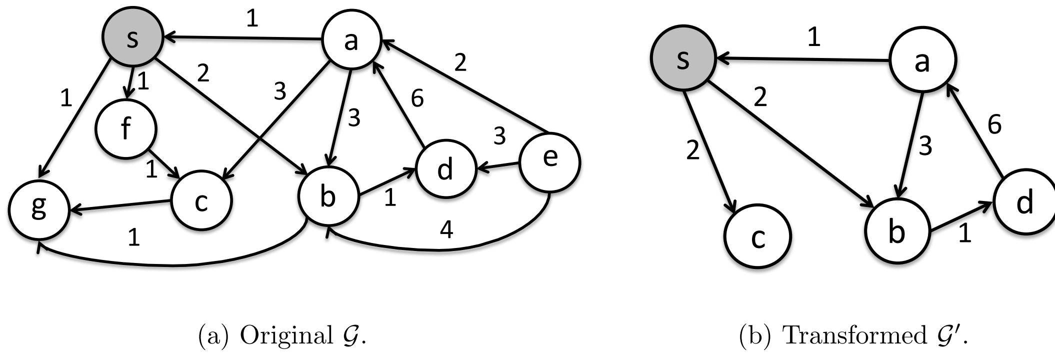
Algorithm 3: SSSP Algorithm.

```
1  Function TwoPhaseSSSP(inputG,srcVertex )
2  |  ▷ V_G of a vertex v = length of the shortest path from srcVertex to v
3  Function ReduceGraph(G, T, Δ, srcVertex)
4  |  ▷ srcVertex is not part of applied T's
5  Function InitializeSSSP(input G; srcVertex)
6  |  ▷ Initialize V_G
7  |  for ( Vertex v : G ) do
8  |  |  V_G[v] ← ∞
9  |  V_G[srcVertex] ← 0
10 |  ▷ Initialize WorkQ
11 |  WorkQ.add( outNeighbors(srcVertex) )
12 Function UpdateVals (v, V_G)
13 |  Updated ← false
14 |  for ( Vertex v' : inNeighbors (v) ) do
15 |  |  if (V_G[v] > V_G[v'] + wt(v', v)) then
16 |  |  |  V_G[v] ← V_G[v'] + wt(v', v)
17 |  |  |  Updated ← true
18 |  return Updated
19 Function Phase2SSSP(V_G', G)
20 |  inival() assigns ∞ or results from phase 1
```

4.1.3 Example: Single Source Shortest Paths

Algorithm 3 presents the two-phased version of the Single Source Shortest Paths (SSSP) algorithm. Only the code sequences that are specific to SSSP are shown while other code sequences from Algorithm 2 remain the same. The function `UpdateVals()` computes the shortest path for a vertex v based on its incoming edges.

Figure 4.1 illustrates graph reduction by converting $\mathcal{G}$ to $\mathcal{G}'$ and Figure 4.2 illustrates how the two-phased SSSP algorithm works on the example graph by first computing $V_{\mathcal{G}'}$ (Figure 4.2-a), then feeding these computed results to $V_{\mathcal{G}}^1$ (Figure 4.2-b), and then computing $V_{\mathcal{G}}^2$ (Figure 4.2-c). In this case a single application of `UpdateVals` in the second phase



(a) Original $\mathcal{G}$.

(b) Transformed $\mathcal{G}'$.

Figure 4.1: Graph reduction example for multiple representation of data technique.

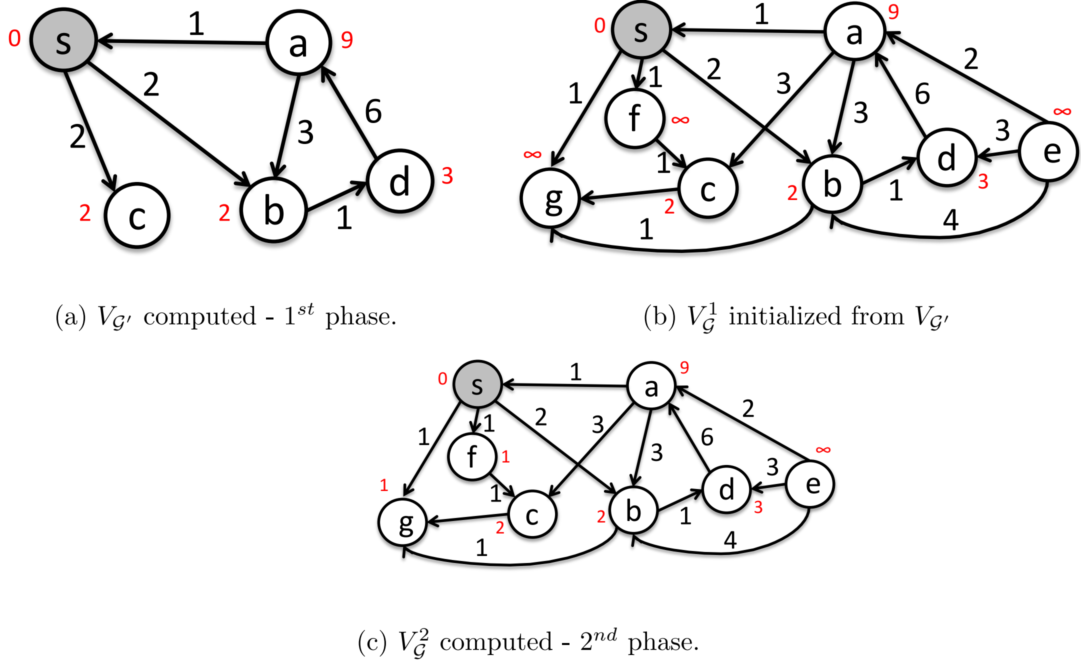
yields precise results (i.e., $V_{\mathcal{G}}^{2} = V_{\mathcal{G}}$). In general, for large complex graphs and different applications, this may not be the case; however, the results computed in the first phase will accelerate the second phase.

4.2 Input Reduction

We present six transformations to reduce input graph and discuss their properties to gain useful programming insights.

4.2.1 Transformations for Input Reduction

Since many graph algorithms are super-linear in the number of edges, the goal of graph reduction is to reduce the number of edges in the graph. If all edges involving a node are eliminated, then so is the node. Figure 4.3 shows the transformations. The red dashed edges are the ones that are eliminated by the transformations. Algorithm 4 presents the algorithm which examines every vertex of the input graph ($\mathcal{G}$), and considers applicability of transformations.



(a) $V_{\mathcal{G}'}$ computed - 1st phase.

(b) $V^1_{\mathcal{G}}$ initialized from $V_{\mathcal{G}'}$

(c) $V^2_{\mathcal{G}}$ computed - 2nd phase.

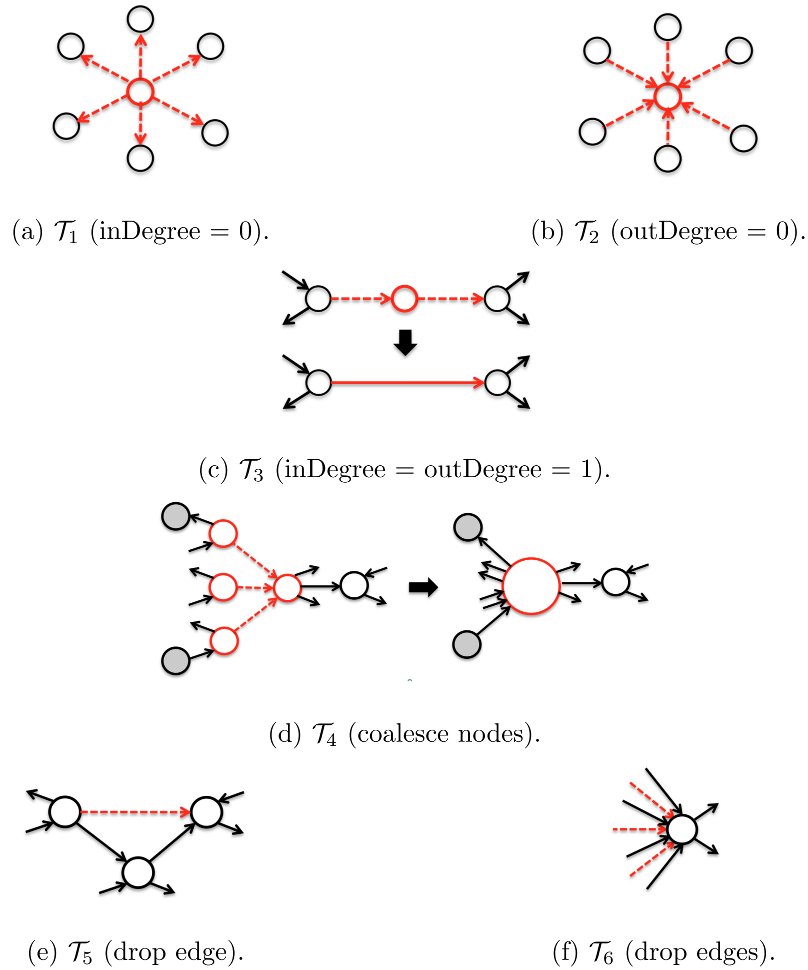
Figure 4.2: Two-phased SSSP processing on $\mathcal{G}$ & $\mathcal{G}'$.

$\mathcal{T}_1/\mathcal{T}_2$. If vertex v has no incoming/outgoing edges, its outgoing/incoming edges are removed and v is dropped.

$\mathcal{T}_3$. For every vertex v with a single incoming and a single outgoing edge, transformation $\mathcal{T}_3$ eliminates v and adds a direct edge between the other end vertices of v's edges. Thus, in a single step we bypass multiple nodes; however, for simplicity we consider bypassing a single node only. Note that $\mathcal{T}_3$ ensures that a path between two vertices v and w is preserved even though direct edges or intervening nodes are dropped.

$\mathcal{T}_4$. For a vertex v with high number of incoming edges,[1] transformation $\mathcal{T}_4$ merges the vertices for those incoming edges with v. $\mathcal{T}_4$ achieves input graph reduction by coalescing directly connected nodes so that the edges connecting them are eliminated

[1] An indegree threshold can be set while using $\mathcal{T}_4$ and $\mathcal{T}_6$. Based on our experiments, we set this threshold to 1,000.



(a) $\mathcal{T}_1$ (inDegree = 0). (b) $\mathcal{T}_2$ (outDegree = 0).

(c) $\mathcal{T}_3$ (inDegree = outDegree = 1).

(d) $\mathcal{T}_4$ (coalesce nodes).

(e) $\mathcal{T}_5$ (drop edge). (f) $\mathcal{T}_6$ (drop edges).

Figure 4.3: Transformations for Input Reduction.

and a reduced graph with fewer edges is obtained. This approach does not reduce connectivity, rather it can introduce new directed paths that were not present in the original graph, hence increasing connectivity. As seen in Figure 4.3, $\mathcal{T}_4$ adds a path between the two gray vertices which is not present in the original graph.

$\mathcal{T}_5$. This transformation drops edge $v \rightarrow w$, if there exists a u such that $v \rightarrow u$ and $u \rightarrow w$. Effectively, for vertex v, $\mathcal{T}_3$ drops the outgoing edge $v \rightarrow w$ if a neighboring vertex of v is directly connected to w. As in $\mathcal{T}_3$, $\mathcal{T}_5$ ensures path preservation; however, $\mathcal{T}_5$

increases the hops/distance between connected vertices.

$\mathcal{T}_6$. Transformations $\mathcal{T}_1$–$\mathcal{T}_5$ can only be applied when their preconditions are satisfied. Thus, the amount of reduction obtained will depend upon the input graph's structural characteristics. In fact, in our experiments the input graph FT is greatly reduced by $\mathcal{T}_1$–$\mathcal{T}_5$ compared to the other graphs. Hence, we introduce transformation $\mathcal{T}_6$ which randomly eliminates incoming edges for a given vertex with high indegree. In this case, the edges are dropped in proportion to the vertex's indegree. Since $\mathcal{T}_6$ can aggressively eliminate edges, it is applied when none of the previous transformations ($\mathcal{T}_1$–$\mathcal{T}_5$) can be used because the vertex does not satisfy their corresponding preconditions.

4.2.2 Transformation Properties

We consider each transformation and deduce strong guarantees about various properties of the transformed graph $\mathcal{G}'$ compared to that of the original graph $\mathcal{G}$. These guarantees are categorized into two types: a) *Structural Guarantees* that determine a relation of structural properties, i.e., *edges*, *vertices* and *components*; and b) *Non-Structural Guarantees* that determine a relation of edge-weights.

Structural guarantees. Consider six transformational properties that determine the relation of structural properties of $\mathcal{G}'$ with $\mathcal{G}$ when transformation $\mathcal{T}_k$ $(1 \leq k \leq 6)$ is applied.

[V-ADD]: $\mathcal{T}_k$ results in vertex v s.t. $v \in \mathcal{G}', v \notin \mathcal{G}$.

[V-SUB]: $\mathcal{T}_k$ results in vertex v s.t. $v \in \mathcal{G}, v \notin \mathcal{G}'$.

[E-ADD]: $\mathcal{T}_k$ results in edge e s.t. $e \in \mathcal{G}', e \notin \mathcal{G}$.

[E-SUB]: $\mathcal{T}_k$ results in vertex e s.t. $e \in \mathcal{G}, e \notin \mathcal{G}'$.

Algorithm 4: Graph Reduction Algorithm.

```
1  Algorithm TRANSFORM( G(V,E) )
2      E′ ← E
3      for ∀v ∈ V do
4          if inDegree(v) = 0 then
5              ▷ apply T_1 : drop v → ∗
6                  E′ ← E′\ outEdges(v)
7          else if outDegree(v) = 0 then
8              ▷ apply T_2 : drop ∗ → v
9                  E′ ← E′\ inEdges(v)
10         else if inDegree(v) = outDegree(v) = 1 then
11             ▷ apply T_3 : bypass v
12                 E′ ← (E′ \ {u → v, v → w}) ∪ {u → w}
13         else if all inNeighbors(v) are unchanged then
14             ▷ apply T_4 : coalesce v and inNeighbors(v)
15                 E′ ← coalesce(G, E′, v)
16     if G requires further reduction then
17         for ∀v ∈ V s.t. v is unchanged do
18             if w ∈ outNeighbors(v) s.t. w is unchanged and outNeighbors(v)
                ∩ inNeighbors(w) ≠ ϕ then
19                 ▷ apply T_5 : drop v → w
20                     E′ ← E′ \ R where R ⊆ inEdges(v)
21     return E′ of G′
22 Algorithm COALESCE ( G(V,E), E′, v )
23     for ∀(w → v) ∈ inEdges(v) do
24         E′ ← E′ \ {w → v}
25         for ∀(u → w) ∈ inEdges(w) do
26             E′ ← E′ \ {u → w}
27             E′ ← E′ ∪ {u → v}
28         for ∀(w → u) ∈ outEdges(w) do
29             E′ ← E′ \ {w → u}
30             E′ ← E′ ∪ {v → u}
31     return E′ of G′
```

[C-MERGE]: $\mathcal{T}_k$ results in a new component c s.t.

$$c_1 \in \mathcal{G}, c_2 \in \mathcal{G}, c = c_1 \cup c_2, c \in \mathcal{G}'.$$

[C-SPLIT] $\mathcal{T}_k$ results in new components c_1 and c_2 s.t.

Trans.	[V-ADD]	[V-SUB]	[E-ADD]	[E-SUB]	[C-MERGE]	[C-SPLIT]
$\mathcal{T}_1$	✗	✓	✗	✓	✗	?
$\mathcal{T}_2$	✗	✓	✗	✓	✗	?
$\mathcal{T}_3$	✗	✓	✓	✓	✗	✗
$\mathcal{T}_4$	✗	✓	✓	✓	✗	✗
$\mathcal{T}_5$	✗	✗	✗	✓	✗	✗
$\mathcal{T}_6$	✗	✗	✗	✓	✗	?

Table 4.1: Structural guarantees for each transformation. ✓and ✗ indicate occurrence and non-occurrence of the corresponding property respectively, whereas ? indicates that the corresponding property may or may not occur.

$$c_1 \in \mathcal{G}', c_2 \in \mathcal{G}', c = c_1 \cup c_2, c \in \mathcal{G}.$$

It is easy to follow that $\mathcal{T}_1$ and $\mathcal{T}_2$ guarantee occurrence of [V-SUB], [E-SUB] and non-occurrence of [V-ADD], [E-ADD], [C-MERGE]. Also, [C-SPLIT] can occur when these two transformations are applied. Transformations $\mathcal{T}_3$ and $\mathcal{T}_5$ guarantee occurrence of [E-SUB] and non-occurrence of [V-ADD], [C-MERGE], [C-SPLIT]. $\mathcal{T}_3$ also guarantees occurrence of [E-ADD] and [V-SUB], whereas $\mathcal{T}_5$ also guarantees non-occurrence of [E-ADD] and [V-SUB]. Transformation $\mathcal{T}_4$ guarantees occurrence of [V-SUB], [E-ADD], [E-SUB] and non-occurrence of [V-ADD], [C-MERGE], [C-SPLIT]. Finally, $\mathcal{T}_6$ guarantees occurrence of [E-SUB] and non-occurrence of [V-ADD], [V-SUB], [E-ADD], [C-MERGE]. While dropping edges using $\mathcal{T}_6$, [C-SPLIT] can occur.

Table 4.1 overviews all structural properties guaranteed by each of the transformations. Note that all transformations guarantee non-occurrence of [V-ADD] and occurrence of [E-SUB] which result in reduction of transformed graph sizes.

Non-Structural guarantees. Since transformations $\mathcal{T}_3$ and $\mathcal{T}_4$ guarantee occurrence of [E-ADD], correct edge weights need to be assigned to newly added edges for weighted graphs. We define two transformational properties which determine the relation of edge weights of $\mathcal{G}'$ with that of $\mathcal{G}$ when transformation $\mathcal{T}_k$ $(1 \leq k \leq 6)$ is applied. In the following expressions, $a \implies b$ means $b \in \mathcal{G}'$ is resulted from $a \in \mathcal{G}$.

[E-EQUAL] $\mathcal{T}_k$ results in edges e_1 and e_2, both with weights $w(e)$ s.t. $e_1 \in \mathcal{G}, e_1 \notin \mathcal{G}', e_2 \in \mathcal{G}', e_2 \notin \mathcal{G}, e_1 \implies e_2$.

[E-FUNC] $\mathcal{T}_k$ results in edges e_1, e_2 and e_3, with weights $w(e_1)$, $w(e_2)$ and $w(e_3)$ respectively s.t.

$\{e_1, e_2\} \in \mathcal{G}, \{e_1, e_2\} \notin \mathcal{G}', e_3 \in \mathcal{G}', e_3 \notin \mathcal{G},$

$w(e_3) = func(w(e_1), w(e_2)), (e_1, e2) \implies e_3.$

[E-FUNC] represents the weight of the newly added edge as a function of weights of edges from the original graph that resulted in this new edge. For example, the new weight can be set as the *sum*, *minimum*, or *maximum* of the original edge weights ([E-SUM], [E-MIN], or [E-MAX] respectively).

Transformation $\mathcal{T}_3$ guarantees occurrence of [E-FUNC] and non-occurrence of [E-EQUAL]. For transformation $\mathcal{T}_4$, both [E-EQUAL] and [E-FUNC] can occur. As we will see in Section 4.3.1, we use [E-SUM] to benefit the exploratory and traversal based graph algorithms.

4.3 Programming for Transformed Graphs

Using the transformations described in Section 4.2.2, we discuss properties of vertex-centric graph algorithms that permit them to benefit from the two-phased model.

4.3.1 Impact of Transformations on Vertex Functions

Since the aforementioned transformations change the structural and non-structural properties of the graph, it is important to determine the impact of these changes on how programmers should correctly express graph algorithms. Even though custom algorithms can be written so that computations performed on transformed graphs always lead to correct values, we eliminate this programming overhead by supporting the popular vertex centric programming for our two-phased processing model.

Vertex-centric programming. In this model, algorithms are expressed in a vertex-centric manner, i.e., computations are written from the perspective of a single vertex. These computations, called *vertex functions*, are iteratively executed on all vertices, until all the vertex values in the graph stabilize. Vertex functions typically use the values coming from its incoming edges as inputs for computation. Hence, the newly computed value of a vertex depends on the values coming from its incoming edges. Moreover, the asynchronous nature of the graph algorithms requires computations over updates coming from incoming edges to be *commutative* and *associative* — this way, updates coming from different incoming edges can be processed in any order, e.g., the order of their arrival.

To guarantee correct answers at the end of computation, we need to reason about the behavior of vertex functions, first when applied on the transformed graph $\mathcal{G}'$, and later

Algorithm 5: Variants of SSSP vertex functions.

```
1  Function SSSP-IN(Vertex v)
2      if ( v = source ) then
3          return 0
4      minPath ← ∞
5      for ( Vertex u : inNeighbors (v) ) do
6          if ( u.path + wt(u, w) < minPath ) then
7              minPath ← u.path + wt(u, w)
8      return minPath
9  Function SSSP-SIN(Vertex v)
10     if ( v = source ) then
11         return 0
12     minPath ← v.path
13     for ( Vertex u : inNeighbors (v) ) do
14         if ( u.path + wt(u, w) < minPath ) then
15             minPath ← u.path + wt(u, w)
16     return minPath
```

on the original graph $\mathcal{G}$. For illustration, we use two versions of the SSSP vertex functions, SSSP-IN and SSSP-SIN, shown in Algorithm 5. Computations in SSSP-IN only depend on values coming from incoming neighbors, whereas those in SSSP-SIN depend on the previous value of the vertex in addition to the values coming from neighbors. The only difference between SSSP-IN and SSSP-SIN is the initialization of *minPath* (line 3 and 14 marked in red); the rest of the functions are identical. Note that both of these variants produce correct results when used in the traditional vertex centric processing model. However, they behave differently when used in our two-phased processing model, in which only SSSP-IN leads to accurate results.

Let us evaluate each of the structural and non-structural properties which are affected by our transformations.

(A) [V-SUB] and [E-SUB]: [E-SUB] leads to computations being performed even when all the incoming edges of a vertex are not available. Such computations are equivalent to that in the staleness-based (i.e., relaxed consistency) computation model [84] where the edges can potentially contain stale values; in this case, missing edges can be viewed as edges with no new contribution. The same argument also holds true for [V-SUB] since the effect of vertex deletion is viewed as edge deletion by its neighbors, reducing to [E-SUB]. In both of these cases, SSSP-IN and SSSP-SIN produce an over-approximation of path distance when applied on $\mathcal{G}'$, compared to the precise distance computed on $\mathcal{G}$, i.e., $minPath(\mathcal{G}') \geq minPath(\mathcal{G})$. In the second phase when missing vertices and edges become available in $\mathcal{G}$, this approximation automatically gets corrected.

(B) [E-ADD], [E-EQUAL], and [E-FUNC]: Transformations resulting in [E-ADD] are introduced in order to preserve the connectivity in the graph which is essential for various traversal-based graph algorithms. Moreover, both [E-EQUAL] and [E-SUM] attempt to create edge-weights of newly added edges to represent an approximation of the distance between corresponding vertices in the original graph. This allows traversal algorithms to proceed with computations based on those newly added edges since the results for transformed graphs are close to the results for the original graph, and hence can accelerate processing over the original graph in the second phase. However, care must be taken to ensure that algorithms which cannot tolerate such newly added relationships do run correctly; in such cases, the newly added edges can be eliminated dynamically from the computation. When [E-ADD] results from eliminating intermediate vertices such that

there is a path between the end vertices in $\mathcal{G}$ (as in $\mathcal{T}_3$), correctness of both SSSP-IN and SSSP-SIN is guaranteed by [E-SUM].

However, $\mathcal{T}_4$, which results in [E-EQUAL], can add an edge between two vertices across which a directed path did not exist in $\mathcal{G}$. In this case, the approximation computed by SSSP-IN and SSSP-SIN can include calculated paths that are smaller than the true shortest paths. During the second phase using $\mathcal{G}$, SSSP-IN recovers from such an approximation since the computation of a path does not depend on its own previous value, resulting in 100% accurate results.[2] On the other hand, computation in SSSP-SIN relies on the previously computed path value for the given vertex, and hence SSSP-SIN cannot recover from an approximate solution. In this case, instead of directly using [E-EQUAL], the edge weight for such newly added edges resulting in new paths can be set to ∞ (**[E-INF]**) which can guarantee 100% accurate results for SSSP-SIN as well.

(C) [C-SPLIT]: Finally, transformations resulting in [C-SPLIT] typically do not impact correctness since computations are performed locally at vertex-level. If the algorithm requires collaborative tasks at component level, they can be performed correctly in the second phase on the original graph. In our examples, both SSSP-IN and SSSP-SIN remain unaffected by [C-SPLIT].

Transformations beyond $\mathcal{T}_1$–$\mathcal{T}_6$. Note that our transformations can be used as fundamental building blocks to create more complicated transformations which can be applied to reduce the graph size. Conversely, the correctness of graph algorithms while using any new transformation $\mathcal{T}_x$ ($x > 6$) can be argued by reducing the new transformation to one

[2]This is true for graph structures consisting of loops as well.

or many of the proposed set of transformations. If there exists a sequence of transformations among $\mathcal{T}_1$–$\mathcal{T}_6$ which produces the same transformed subgraph as that produced by $\mathcal{T}_x$, correct answers can be guaranteed at the end of computation using the transformed graph produced by $\mathcal{T}_x$. For some $\mathcal{T}_x$ which cannot be expressed as a sequence of proposed transformations, arguments using their structural and non-structural properties can be used to ensure correctness of results. Note that this relationship is *transitive* and hence, the newly proved $\mathcal{T}_x$ can be further used along with $\mathcal{T}_1$–$\mathcal{T}_6$ to prove correctness of results while using other new transformations.

Algorithm	**Vertex Function**
SSSP	$v.path \leftarrow \min_{e \in \text{inEdges}(v)} (e.source.path + e.wt)$
SSWP	$v.path \leftarrow \max_{e \in \text{inEdges}(v)} (\min(e.source.path, e.wt))$
CC	$v.component \leftarrow \min_{e \in \text{edges}(v)} (e.other.component)$
GC	$change \leftarrow \bigvee_{e \in \text{edges}(v)} (v.color == e.other.color)$ **if** $change == true$ **then:** $v.color \leftarrow c$: where $\forall_{e \in \text{edges}(v)}(e.other.color \neq c)$
PR	$v.rank \leftarrow 0.15 + 0.85 \times \sum_{e \in \text{inEdges}(v)} e.source.rank$
CD	$\forall_{e \in \text{edges}(v)} frequency[e.other.community] \mathrel{+}= 1$ $v.community \leftarrow c$: where $frequency[c] = \max_{i \in frequency} (frequency[i])$

Table 4.2: Various vertex-centric graph algorithms. SSSP, SSWP, CC, PR, and GC produce 100% accurate results.

4.3.2 Graph Algorithms

We now discuss each of the graph algorithms used in this work. Table 4.2 shows details of vertex functions. We will argue that PR, SSSP, SSWP, GC, and CC produce 100% accurate results whereas the same accuracy cannot be ensured by CD.

(A) Shortest & Widest Paths: As discussed in Section 4.3.1, when shortest path (SSSP) is computed on $\mathcal{G}'$, the transformations lead to an approximate solution which gets corrected in the second phase of processing when using SSSP-IN. For the widest path (SSWP), recall that [E-SUM] is a specialization of [E-FUNC] which can support a wide range of such traversal based algorithms. Hence, SSWP can be supported by ensuring that the weight of any newly added edge is the minimum of the edges whose removal caused the addition of this new edge ([E-MIN]). In this case, [E-MIN] ensures that the calculated path width in $\mathcal{G}'$ is always at most that of the equivalent path in $\mathcal{G}$.

(B) Connected Components: Since the main idea behind CC is that vertex values within a component are the same and those in different components are different, we determine its correctness using [C-MERGE] and [C-SPLIT] properties. All the transformations guarantee non-occurrence of [C-MERGE]; hence, values flowing in different components of the original graph will always be different in the transformed graph. When [C-SPLIT] occurs, vertices within the same component of the original graph can now belong to different components of the transformed graph, leading to different values flowing in the same original component. This approximation gets corrected when these vertices are re-grouped together into the same component in the second phase; the computation simply picks one of the vertex values to flow across the entire component.

(C) Graph Coloring: The underlying idea behind GC is to assign different colors to the end vertices of every edge while using minimal [3] set of colors to color all vertices. Hence, we determine its correctness using [E-ADD] and [E-SUB] properties. When [E-ADD] occurs, an edge connects two vertices in $\mathcal{G}'$, which were disconnected in $\mathcal{G}$. Even though this causes the two vertices to be assigned different colors, it does not violate the correctness of the solution: when the edge is removed in the second phase, the color assignment for one of these two vertices gets updated and is propagated throughout the graph. When [E-SUB] occurs, vertices which are connected by an edge in $\mathcal{G}$ become disconnected. This can cause the vertices to be assigned the same color when processing on $\mathcal{G}'$. However, during the second phase, these edges become available in $\mathcal{G}$ which re-processes the vertices and hence, the self-correcting nature of the algorithm detects and corrects the coloring inconsistency. This in turn ensures that different colors are assigned to connected vertices. Note that different executions of the same original graph coloring algorithm on the same graph can result in different color assignments and minimal number of colors, i.e., the set of correct solutions is not a singleton and hence, the solution computed by our two-phased approach is one of the solutions in the correct set because it adheres to the two constraints of the problem.

(D) PageRank: As shown in [21], PR converges to the correct solution regardless of the initial vertex values. With different initializations, the path to convergence changes. Since computations over $\mathcal{G}'$ provide an approximation of the final results, these results, when fed as initialization values for $\mathcal{G}$, cause the second phase to converge faster.

[3] Graph coloring is NP-complete and hence the constraint is usually relaxed to minimal colors which can be solved in polynomial time.

(E) Community Detection: CD detects communities in the graph by propagating labels that are most frequent among the immediate neighborhood of the vertices. Both [E-SUB] and [E-ADD] influence this computation since the frequency of labels get affected by edge addition/deletion, which leads to an approximation at the end of first phase. During the second phase when $\mathcal{G}$ becomes available, this approximation may not be fully corrected because individual corrections due to availability of original edges might not affect the highly approximate frequency calculated in previous iterations. This can lead to results which are not accurate.

Early termination in the first phase. A key advantage of our approach is that none of the algorithms require processing over $\mathcal{G}'$ to converge to its final solution before moving on to $\mathcal{G}$. This is because the intermediate values produced while processing $\mathcal{G}'$ also represent a valid approximation of the final solution. Hence, to speed up the computation even further, we can employ *early termination* of first phase, where the computation does not wait to reach to its converged solution, and the available computed values are directly used in the second phase to process the original graph.

4.4 Analysis and Generality

We first theoretically analyze the performance benefits that can be achieved by our two-phased model and then discuss the generality of our approach to achieve similar benefits in different scenarios.

4.4.1 Analysis

Let P_G and P_T be the average execution times of a single iteration over G (original graph) and G_T (reduced graph) respectively. Further, let P_G^T be the average execution time of a single iteration over G in the second phase using computed results fed from G_T to G. Note that $P_G^T < P_G$. Moreover, since $|G_T| < |G|$, i.e., G_T has fewer edges than G, we know that $P_T < P_G$. In order to accelerate processing using the two-phased approach, we require:

$$I_1.P_T + I_2.P_G^T < I.P_G \tag{4.1}$$

where I_1, I_2, and I are the number of iterations in which G_T is processed in the first phase, G is processed in the second phase when computed results are fed from G_T, and G is processed in the original processing model, respectively. Upon rearranging Equation 4.1 we get:

$$I_1.P_T < I.P_G - I_2.P_G^T \tag{4.2}$$

which conveys that in order to achieve benefits from our technique, the savings from the second phase ($I.P_G - I_2.P_G^T$) should be larger than the time spent in the first phase ($I_1.P_T$).

For example, if we want to accelerate the overall processing by 25%, we should have:

$$\begin{aligned}
& I_1.P_T + \frac{1}{4}.I.P_G = I.P_G - I_2.P_G^T \\
\implies \quad & I_1.P_T = \frac{3}{4}.I.P_G - I_2.P_G^T \\
\implies \quad & I_1.P_T < \frac{3}{4}.I.P_G \quad \rightsquigarrow \quad |G_T| < \frac{3}{4}.|G|
\end{aligned} \tag{4.3}$$

The above implication from processing times to graph sizes ($|G|$ and $|G_T|$) is an approximation

that holds true as G_T is created primarily by dropping vertices and edges from G and hence, $I_1.P_T$ reduces proportionately compared to $I.P_G$.

Equation 4.3 shows that if we want to accelerate the overall processing by 25% using our two-phased processing technique, we must ensure that the reduced graph is reduced to at least 75% of the original graph.

4.4.2 Generality

From the above analysis, it can be clearly seen that the savings in the overall processing times are largely dependent on $|G|$ and $|G_T|$, i.e., size of original and transformed graphs. This allows us to argue that the our technique is independent of the underlying processing environments, iterative algorithms, and input graphs.

Processing environments. Processing large graphs in different environments incurs different overheads and since our technique eliminates significant amount of processing on the entire large graph, it can help alleviate some of these overheads. For example, processing large graphs on GPUs would require frequent transfer of subgraph information and computed values between host-memory and device-memory which is a significant overhead [37, 71]. Since our transformed graph is much smaller, the bulk of this transfer gets eliminated in the first phase and is only performed for remaining few iterations in the second phase. Moreover, if the transformed graph fully fits in the GPU memory, absolutely no transfers are required in the first phase.

In a distributed processing environment, the overall performance is largely dependent on the communication of vertex updates between nodes [84]. Again, using our

technique, much of the communication can be avoided in the first phase, hence reducing the overall communication overheads. Moreover, the transformed graph in the first phase can be processed on the subset of nodes in the cluster to reduce synchronization and communication overheads.

The applicability is similar in an out-of-core processing environment where the graph is resident on secondary storage [44, 66]. The first phase eliminates costly disk read and writes while reducing them in the second phase due to reduction in number of iterations.

Iterative algorithms. The two-phased processing is suitable for iterative graph algorithms whose convergence is dependent on the values being computed. The performance benefits are noteworthy for different kinds of graph algorithms: on one hand, traversal algorithms like SSSP/SSWP which require less computation and on other hand, algorithms like PR/GC/CD which require more computation to compute the final solution. Also, the benefits achieved are higher for asynchronous graph algorithms [84] because correctness guarantees are stronger for those cases. Again, as deduced in the above analysis, the performance benefits of our technique are mainly due to reduction in the data-size that needs to be processed and is independent of the kind of processing being performed on the data.

Input graphs. The proposed reduction and processing techniques are best suited for irregular graphs where the degree distribution across vertices is spread across a wider range, allowing various pre-conditions for our transformations to be satisfied.[4] As long as the input graph is large enough that a reduction in its size achieves perceivable reduction in

[4]Real-word graphs from various domains like social network analytics, web analytics, mining, etc. are highly irregular.

processing time, the two-phased processing model can be used to accelerate processing. The large real-world input graphs which are highly irregular and sparse for our evaluation in Chapters 5 and 6 on which our technique achieves reasonable benefits. Moreover, our transformations $\mathcal{T}_4$ and $\mathcal{T}_6$ are tunable so that they can be applied even to a graph on which no other transformations can be applied.

4.5 Summary

In this chapter, we presented a generalized technique for accelerating graph applications. The novel technique presented in this chapter executes the application in two phases while using multiple input representations: first, a smaller representative graph is processed by the application until convergence; second, this output is used for computing the final result with original graph as the input. Next, we demonstrate the applicability of the two-phased technique on parallel graph processing environment (Chapter 5). In Chapter 6 we present a modified version of the two-phased technique for accelerating out-of-core graph applications.

Chapter 5

Employing Multiple Data Representations for Multithreaded Graph Processing

Programmers often use parallel processing to speed up computations over a large graph. With the availability of multi-core and many-core machines, these parallel executions have gained popularity. Several domain specific languages and programming models for parallel graph analytics have been proposed which simplifies the task for expressing the graph algorithm. The typical parallel processing of the graph requires the entire graph to be stored in memory. Many popular parallel graph applications are iterative in nature, which requires accessing the entire graph multiple times. This behavior makes the parallel graph application a perfect example where two-phased processing could accelerate the execution. In this chapter, we present a general *parallel vertex-centric graph* algorithm and its two-phased

version (Section 5.1). We evaluate our proposed technique on six benchmark applications using real-world graphs which are highly irregular and sparse (Section 5.2).

Algorithm 6: Parallel Iterative Vertex-Centric Graph Algorithm.

```
1  Function iA (input 𝒢)
2      Initialize V_𝒢 & WorkL
3      VDist ← DistributeVertex(𝒢,NUMTHREADS)
4      while ( ! WorkL.empty ) do
5          par-for ( i : NUMTHREADS )
6              for ( v : VDist[i] ) do
7                  if ( WorkL.contains(v) ) then
8                      WorkL.remove(v);
9                      if ( UpdateVals(v, V_𝒢)) then
10                         WorkL.add( outNeighbors (v) )
11                     end
12                 end
13             end
14         end
15     end
16     return V_𝒢
17 end
18 Function UpdateVals(v, V_𝒢)
19     Updated ← false
20     if ( updateCheck(v, inNeighbors(v) ) then
21         update V_𝒢[v]
22         Updated ← true
23     end
24     return Updated
25 end
```

5.1 Parallel Original and Two-Phased Algorithms

This section provides an overview of the application of the two-phased method on a parallelized version of the vertex-centric graph application. A general form of an original parallel vertex-centric graph algorithm (Algorithm 6) and its corresponding two-phased version are presented (Algorithm 7). The function `iA` in Algorithm 6 represents the

original parallel algorithm whose application to graph $\mathcal{G}$ produces the accurate ($V_{\mathcal{G}}$) result. The DistributeVertex method (line 3) distributes the vertices of the graph $\mathcal{G}$ equally in NUMTHREADS and generates VDist (line 3), which contains vertices to be processed by each thread, i.e., VDist[i] contains the list of vertices to be processed by thread i. The vertices to be processed in future iterations are stored in work list (WorkL). The parallel processing logic is: each thread traverses its vertex list (line 6) and calls the UpdateVals method only if the vertex is present in WorkL. Therefore, multiple threads would access the graph several times per iteration. The two-phased method would greatly benefit graph applications executing in parallel.

In Algorithm 7, we have omitted the methods which are the same as Algorithm 6 for brevity. Function TPiA is the two-phased version that calls iA (Algorithm 6) and iAP2 in first and second phases, where both functions process the graph in-parallel. The ease of programming for the two-phased method lies in the fact that the processing logic for iAP2 in Algorithm 7 (lines 22–33)is exactly the same as that in iA in Algorithm 6 (lines 4–15).

5.2 Evaluation

We thoroughly evaluate our two-phased processing technique to show that our approach is *efficient* (savings in execution time), *scalable* (higher savings in execution with higher number of threads) and produces *accurate* results for most of the graph applications with low *time* overhead.

Benchmarks, Inputs and System. We consider six popular vertex centric graph algorithms, as shown in Table 5.1. We implemented baseline and the two-phased version of each

Algorithm 7: Parallel Two Phase Vertex-Centric Graph Algorithm.

```
1  Function TPiA (input 𝒢)
2      𝒢′ ← ReduceGraph ( 𝒢, 𝒯, Δ )
3      V¹_𝒢 ← iA(𝒢′)
4      V²_𝒢 ← iAP2(V¹_𝒢, 𝒢)
5      return V²_𝒢
6  end
7  Function iAP2(V¹_𝒢, 𝒢)
8      Initialize WorkL
9      VDist ← DistributeVertex(𝒢,NUMTHREADS)
10     par-for ( i : NUMTHREADS )
11         for ( v : VDist[i] ) do
12             if ( v ∈ 𝒢′ ) then
13                 V²_𝒢 ( v ) ← V¹_𝒢 (v)
14             end
15             else
16                 V²_𝒢 ( v ) ← initval ( )
17                 WorkL.add ( v )
18             end
19         end
20     end
21     WorkL.add ( Vertex v s.t. v is affected by addition / deletion of edges)
22     while ( ! WorkL.empty ) do
23         par-for ( i : NUMTHREADS )
24             for ( v : VDist[i] ) do
25                 if ( WorkL.contains(v) ) then
26                     WorkL.remove(v);
27                     if ( UpdateVals(v, V_𝒢)) then
28                         WorkL.add( outNeighbors (v) )
29                     end
30                 end
31             end
32         end
33     end
34     return V_𝒢
35 end
```

of the benchmarks in Galois [62], a state-of-the-art parallel execution framework.

Table 5.2 shows the details of the input graphs, their reduced versions and time taken for reduction. We use 4 input graphs, 3 of which are real-world graphs (Friendster,

Twitter and UKDomain) from the public Konect repository [42]. The synthetic graph (RMAT-24) is a scalefree graph (a = 0.5, b = c = 0.1, d = 0.3) similar to the one used in [62]. To transform these graphs, we define a tunable parameter **Edge Reduction Percentage (ERP)** as:

$$ERP = \frac{|E_{\mathcal{G}'}|}{|E_{\mathcal{G}}|} \times 100$$

where $|E_{\mathcal{G}}|$ and $|E_{\mathcal{G}'}|$ are the number of edges in original graph $\mathcal{G}$ and the reduced graph $\mathcal{G}'$. We generate the reduced graphs with varying ERP (75%, 70%, 60%, 50%, 40% and 30%) using our transformation tool based on Algorithm 4.

Benchmark	**Type**
Single Source Shortest Path (SSSP) Single Source Widest Path (SSWP) PageRank (PR) Graph Coloring (GC) Connected Components (CC)	Accurate
Community Detection (CD)	Approximate

Table 5.1: Graph Algorithms.

Input Graph		**Graph Size**		**Reduction Time (sec)**
		#Nodes	**#Edges**	
Friendster (FT)	Original	68.3M	2.6B	5.63-9.37
	Reduced	41.9-51.8M	0.78-1.9B	
Twitter (TT)	Original	41.7M	1.5B	1.31-7.13
	Reduced	23.4-30.8M	0.4-1.1B	
UKDomain (UK)	Original	39.5M	936.4M	1.31-7.13
	Reduced	27.6-32.1M	280.9-702.3M	
RMAT-24 (RM)	Original	17M	268M	0.05-0.34
	Reduced	11.6-13.5M	80.4-201M	

Table 5.2: Input Graphs.

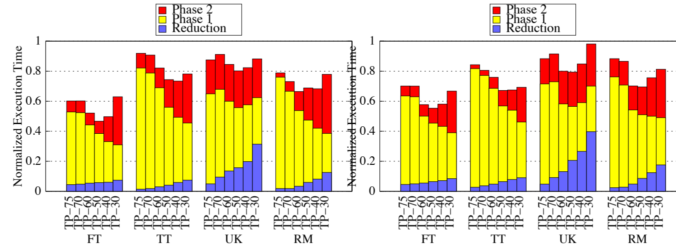
Experiments were performed on a machine with 4 six-core AMD^{TM} 8431 processors (total 24 cores) and 32 GB RAM running Ubuntu 14.04.1 (kernel version 3.19.0-28-generic). The programs were compiled using GCC 4.8.4, optimization level `-O3`.

We evaluate the performance of following versions of the benchmark implementations:

- **Baseline:** based on the traditional processing model.
- **TP-X:** based on our two-phased processing model using reduced graphs with ERP = X%. Note that the execution times include the graph reduction times which are already presented in Table 5.2.

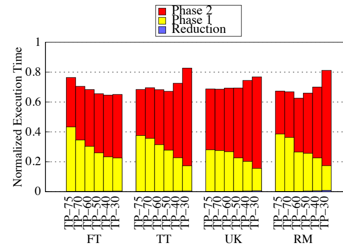
Unless otherwise specified, the benchmarks were run with 20 software threads.

Efficiency of Two Phased Processing. Figure 5.1 shows the speedups achieved by TP-X over Baseline for X ∈ {30%, 40%, 50%, 60%, 70%, 75%}. As we can see, the speedups increase as ERP decreases from 75% to 40%; on average, TP-75, TP-70, TP-60, TP-50 and TP-40 achieve a speedup of 1.16×, 1.20×, 1.3×, 1.37× and 1.29× respectively. This is because of the high savings achieved in the second phase while processing the original graphs. On average for TP-75, TP-70, TP-60, TP-50, and TP-40, the savings achieved in the second phase are 79.26%, 77.30%, 74.16%, 71.09% and 66.88% respectively. These high savings allow tolerating the execution times of reduction and first phase over reduced graphs; the execution times normalized w.r.t. *Baseline* for the first phase of TP-75, TP-70, TP-60, TP-50, and TP-40 are 0.65, 0.59, 0.50, 0.42, and 0.37 seconds, respectively; time taken by reduction is as low as 0.01, 0.02, 0.03, 0.03, and 0.04 seconds, respectively. Since

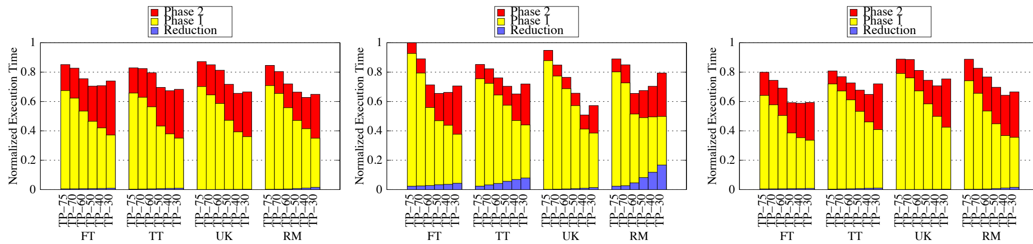


(a) Normalized Execution times for SSSP. For comparison, the Baseline execution times (in sec) for FT, TT, UK and RM are 127.14, 96.15, 4.65 and 2.71 respectively.

(b) Normalized Execution times for SSWP. For comparison, the Baseline execution times (in sec) for FT, TT, UK and RM are 134.67, 104.32, 4.81 and 2.9 respectively.



(c) Normalized Execution times for PR. For comparison, the Baseline execution times (in sec) for FT, TT, UK and RM are 2957, 2120, 298 and 57.64 respectively.



(d) Normalized Execution times for GC. For comparison, the Baseline execution times (in sec) for FT, TT, UK and RM are 1216, 1014, 771 and 33.51 respectively.

(e) Normalized Execution times for CC. For comparison, the Baseline execution times (in sec) for FT, TT, UK and RM are 264.64, 118.71, 137.6 and 3.05 respectively.

(f) Normalized Execution times for CD. For comparison, the Baseline execution times (in sec) for FT, TT, UK and RM are 1351, 896, 654 and 33.25 respectively.

Figure 5.1: Normalized execution time of two-phased execution for each benchmark-graph-`ERP` value.

our reduction transformations are local and non-interfering, the cost of performing the input reduction is much lower than the savings achieved in processing.

As expected, the time taken to process the reduced graph in the first phase decreases as ERP decreases simply because the work done is typically proportional to the size of graph. On the other hand, the execution time in the second phase increases as ERP decreases. This is mainly because an aggressively reduced graph with lower ERP is structurally less similar to the original graph compared to that reduced with a higher ERP. Hence, the values which are fed from reduced graph with lower ERP require more computation in the second phase in order to reach to maximum possible accuracy for the original graphs.

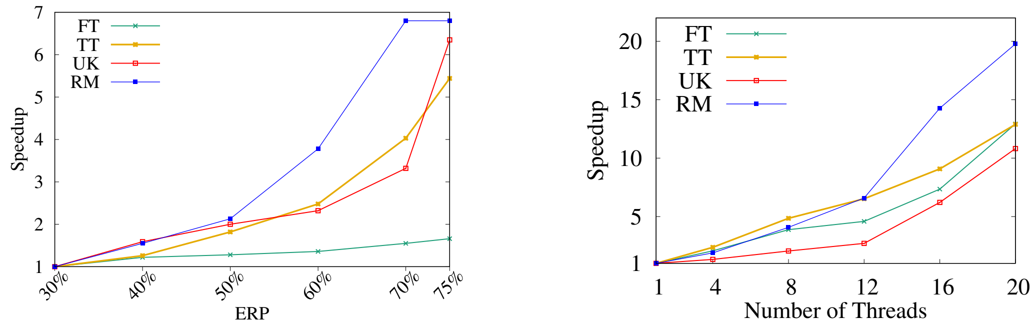


Figure 5.2: Scalability of Reduction Algorithm w.r.t. ERP (left) and number of threads (right) for TP-50.

The savings achieved by our two-phased processing model increases as ERP decreases up to a certain limit. Across each of our benchmark-input-ERP combination, the maximum savings are observed for ERP = 40-50%. However, note that further decreasing ERP reduces the amount of savings achieved; with ERP = 30% the performance degrades and the average speedup drops to 1.41×. This is because the reduced graph with very low ERP becomes too small (i.e., structurally very dissimilar) compared to the original graph

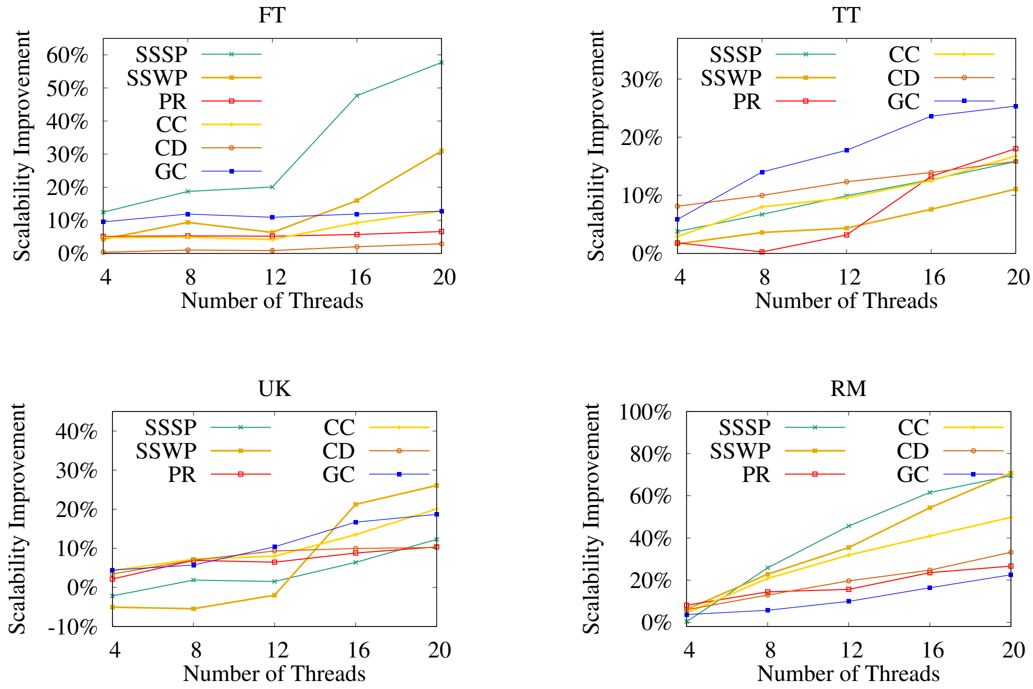


Figure 5.3: Improvement in scalability using the two-phased model with varying number of threads. For comparison, the Baseline execution times (in sec) for PR/SSSP with 1 thread for FT, TT, UK and RM are 24577/1674.43, 22307/1016.88, 3014.39/53.64 and 934.43/12.07.

and the major burden of processing then moves over from first phase to second phase. As an extreme example, one can see that ERP = 0 means that no processing is required in the first phase whereas the second phase is exactly same as processing the original graph from the very beginning.

It is interesting to note that the benefits achieved from our two-phased approach are greater for FT graph (1.37-1.69×) mainly because it is larger than TT and UK graphs.

Scalability of Input Reduction. We study the scalability of our input reduction algorithm while 1) varying ERP from 30% to 75% with 20 threads; and, 2) varying number of

threads from 1 to 20 for TP-50.[1] As we can see in Figure 5.2 (left), with increase in ERP the reduction algorithm performs faster compared to ERP=30% mainly because there are fewer edges to be removed for higher ERP, and hence, the reduction algorithm only needs to traverse certain percentage of the graph to achieve the expected ERP. Moreover, Figure 5.2 shows that the reduction algorithm is scalable w.r.t. the number of threads; this naturally follows from the requirement of the transformations to be local and non-interfering allowing them to be executed at vertex-level in parallel.

Scalability of Two-Phased Processing. As shown in [62], the Baseline system scales well with increase in number of threads. To show the impact of our approach, Figure 5.3 shows the improvement in scalability achieved by TP-50[1] over Baseline while varying the number of threads from 1 to 20. Note that in Figure 5.3 the Baseline is also parallel, i.e., a data-point with t threads represents improvement achieved by our technique using t threads compared to baseline using t threads. As we can see in most cases, the improvements slowly increase as number of threads increase and the maximum improvements are achieved with 20 threads. We believe this is because the reduced graphs become denser compared to the original graphs and hence, the probability of the same vertex to be scheduled multiple times by different threads increases rapidly in TP-50 with increase in threads compared to that in Baseline. This in turn allows more merging of such multiple schedule requests of same vertices to single vertex computations. Moreover, the second phase majorly involves correction of values and hence, lesser contention is expected since probability of all neighboring vertices to be scheduled simultaneously greatly reduces.

[1] Since ERP = 50 performs best across most cases in our previous experiments, we only consider TP-50 to save space.

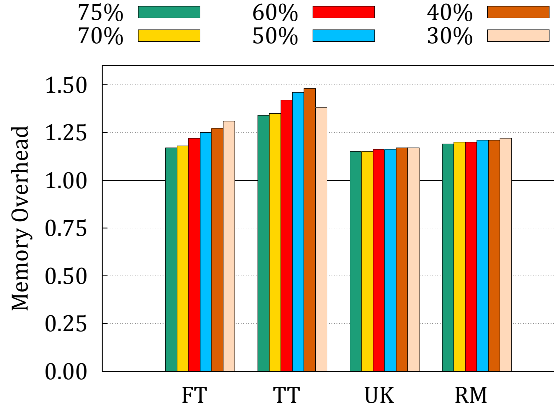


Figure 5.4: Increase in memory footprint.

Memory Overhead. While the two phases can be processed separately, feeding values from the first phase to the next can incur expensive reads and writes which can offset the performance benefits achieved by our technique. Hence, it is crucial to maintain the reduced and original graphs in memory and eliminate the explicit intermediate feeding by incorporating a unified graph which leverages the high structural overlap across the two graphs. Figure 5.4 shows the increase in memory when using a unified graph. On average, the memory consumption increases by 1.25×; it goes higher for TT (1.34×-1.48×) mainly because the percentage of newly added edges in the transformed graphs is much higher (25%-40%) for TT compared to other graphs (2.7%-23%).

Note that the overhead increases as ERP decreases. This is due to the impact of increase in the structural dissimilarity between the original and transformed graphs that requires representing the dissimilar components (i.e., newly added edges) separately for both graphs. Note that these overheads are tolerable compared to those incurred by representing both the graphs separately in memory which can be as high as 1.75×.

Relative Error for CD. As discussed in Section 4.3.2, the accuracy of results for CD could not be guaranteed. In order to determine how good the calculated results are, we define relative error as the ratio of vertices whose computed community values are different compared to the ideal results. Table 5.3 shows the relative error for CD across all input-ERP combinations. As we can see, the relative error is very small; the average relative error across all cases is 0.02 and the maximum relative error is only 0.065. In fact, the relative error for FT across ERP-60, ERP-70 and ERP-75 is very low (<1E-5). It is interesting to note that the error values decrease as ERP increases. This is mainly because with fewer reduction transformations being applied for higher values of ERP, the probability of merging communities in reduced graphs decreases.

Input	TP-30	TP-40	TP-50	TP-60	TP-70	TP-75
FT	0.017	0.002	0.001	<1E-5	<1E-5	<1E-5
TT	0.049	0.041	0.036	0.021	0.019	0.017
UK	0.065	0.023	0.017	0.013	0.012	0.011
RM	0.043	0.034	0.021	0.018	0.012	0.01

Table 5.3: Relative Error for CD.

We further study how the relative error changes during execution by plotting it for TP-40 in Figure 5.5. The vertical dotted lines indicate different phases of execution; the first line (close to 0) indicates the end of reduction process and the second line (in the middle) indicates the end of the first phase and the beginning of the second phase. As we can see, the relative error remains high during the first phase mainly because of the vertices which are missing in the reduced graph.

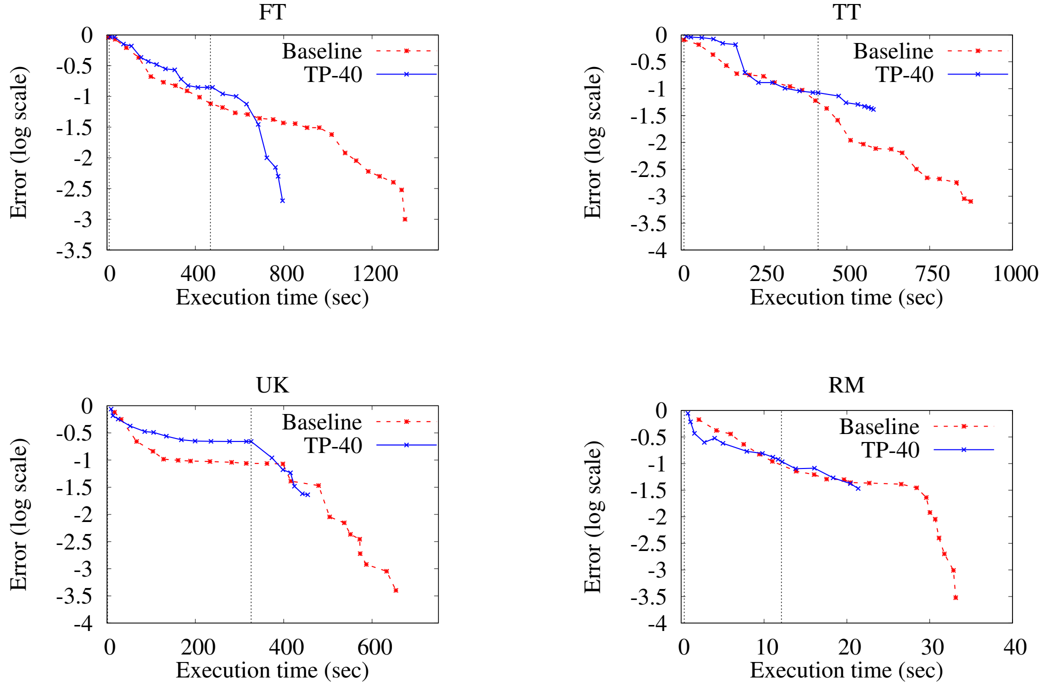


Figure 5.5: Relative Error (log scale) vs. Execution Time (sec) for CD: Baseline and TP-40. Note that the point at which the Baseline version terminates, i.e., relative error becomes zero, is not plotted due to use of log scale.

However, the relative error drops rapidly during the second phase due to availability of missing vertices and edges in the original graph. At the end of the first phase, the relative error for FT, TT, UK and RM remain at 0.014, 0.084, 0.22 and 0.12 respectively.

Contribution of Individual Transformations. Finally, we evaluate the effect of applying individual transformations one after the other on the overall performance. We define a transformation set $\mathcal{T}_{1-k}$ as the set of transformations starting from $\mathcal{T}_1$ up to $\mathcal{T}_k$. Hence, the transformation set $\mathcal{T}_{1-4}$ includes $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$ and $\mathcal{T}_4$ whereas $\mathcal{T}_{1-1}$ only includes $\mathcal{T}_1$.

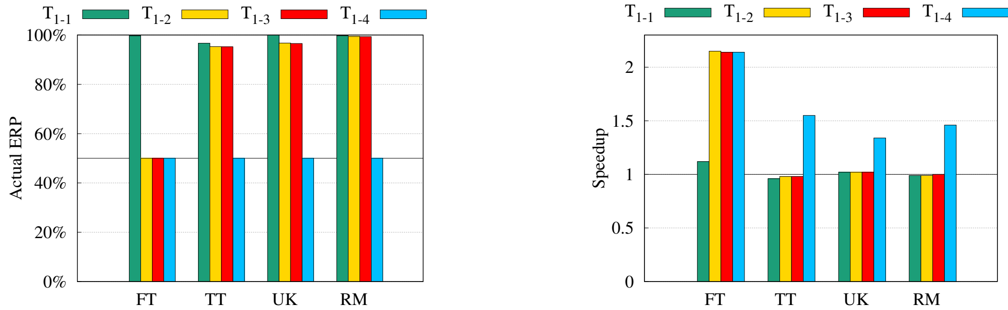


Figure 5.6: Actual ERP achieved (left) and speedups achieved (right) while using different transformation sets when ERP is set to 50%.

The reduced graphs for this set of experiments are generated using different transformation sets $\mathcal{T}_{1-k}$ $(1 \geq k \leq 4)$. To clearly present the impact of transformations on both, the size of reduced graphs and the savings in execution time, we select ERP = 50% and only consider the SSSP benchmark. Figure 5.6 shows the speedups achieved for each of the graphs transformed using the transformation sets, compared to the Baseline. Since the transformations being applied have their pre-conditions which need to be satisfied, the actual ERP using a smaller transformation set can be higher than the requested ERP of 50%. Hence, we also present the actual ERP obtained using the transformation sets.

As we can see in Figure 5.6, $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$ collectively reduce only small portion of TT, UK and RM graphs; $\mathcal{T}_{1-3}$ achieve 95.26%, 96.58% and 99.39% ERP for TT, UK and RM respectively. Due to this, little to no savings are achieved until $\mathcal{T}_4$ is included in the transformation sets for which speedups of up to 1.34–1.55× are achieved. The FT graph, on the other hand, is amenable to $\mathcal{T}_2$ and $\mathcal{T}_3$, allowing 50% ERP to be achieved for $\mathcal{T}_{1-2}$ and $\mathcal{T}_{1-3}$ too. Hence, the speedups achieved for those transformation sets are ∼2.15×.

5.3 Summary

In this chapter, we presented the application of our multiple data representation technique on *parallel* graph processing for accelerating the execution of the iterative algorithms. A detailed evaluation of our proposed execution model on the parallel graph processing shows that speedups up to $2.14\times$ could be achieved. The approach scales well with increase in number of threads, i.e., with increase in number of threads the speed up increases w.r.t parallel baseline. Our technique poses a minimal memory overhead for storing the multiple representations of data (max $1.48\times$). The precise results were obtained for 5 out of 6 benchmarks, with maximum relative error of 0.065. The evaluation was done assuming that the graph fits in memory. In the next chapter, we consider the scenario where the graph does not fit into memory, and propose a modified version of the two-phased processing model for the out-of-core graph applications.

Chapter 6

Employing Multiple Data Representations for Out-of-core Graph Processing

Many modern workloads are large enough to fit in memory (e.g., social networking data and geographical information system), which may run into terabytes in size. Programmers have designed out-of-core applications to handle such large data. In these out-of-core applications, the persistent storage is used as an extension to memory, where the application fetch the parts of input such that it fits in memory. After processing, the results are stored back to disk. The graph applications are iterative in nature, for which the parts of input graph are fetched and the results are stored multiple times, which leads to significant amount of time being spent on disk I/O. The key challenge in these applications is to limit this I/O overhead. The data transformation technique could be extended to reduce the size of

input graph hence reduce I/O. However, there are challenges in direct application of the two-phased processing model: first, the data transformation could not be applied directly (unlike shared-memory approach) since the graph itself could not fit in memory: second, the transformed graph could not possibly fit in memory. The out-of-core execution would be necessary in the first phase, which might subsume any savings in execution time using two-phased execution.

To address these challenges, this chapter provides a modified version of two-phased approach which reduces the disk I/O during execution. It uses a multi-level edge-cut reducing partitioning scheme to partition the graph into smaller subgraphs, then the data transformation technique is applied for creating multiple representation of input graph such that each transformed subgraph fits in memory. It achieves savings in disk I/O time, thereby ensuring the efficient execution under memory limitation.

6.1 Out-of-Core Graph Processing

There are many popular out-of-core graph application frameworks (e.g., Graphchi [44] and X-Stream [66]). The Graphchi framework uses vertex-centric programming model to express the graph applications and, X-Stream uses edge-centric programming model. The data transformation and the two-phased processing model was designed around vertex-centric programming model, hence Graphchi is chosen for the purpose of extension and evaluation.

Next, we turn our focus on the functioning of Graphchi, and present how a graph is processed in an out-of-core fashion. Figure 6.1 presents the high level overview of Graphchi out-of-core processing. The programmer provides the graph processing system with a User-

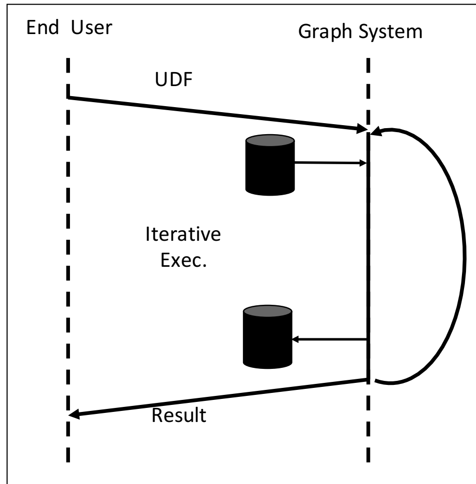


Figure 6.1: High level overview of our approach.

Defined Function (UDF), which is the vertex-centric graph application code. Graphchi process the graph by the following steps: 1) The input graph is read, vertices and edges are partitioned into chunks; 2) Each chunk is loaded from disk to memory and UDF for all the vertices in the corresponding chunks is applied; 3) Computed values are stored back onto disk; 5) Step 2 and 3 are repeated until the computed values converge.

Table 6.1: I/O time analysis for PR and CC graph applications.

App. / **Graph**	**CC**			**PR**		
	Exec. Time (sec)	I/O Time (sec)	I/O %	Exec. Time (sec)	I/O Time (sec)	I/O %
WK_EN	141.81	86.6	61	272.75	190.97	70
WK_FR	43.5	27.61	63	75.88	56.44	74

We studied the time spent on disk I/O for two real-world graph applications: Pagerank (PR) and Connected Components (CC). Each graph application is run with two real-world graphs from Konect [42]: first, Wikipedia Links in English (WK_EN) graph which has 29M vertices and 266M edges; second, Wikipedia Links in French (WK_FR) graph which

ha 3M vertices and 102M edges. We measured execution time and time spent on I/O for three applications and two graphs and present our findings in Table 6.1. The goal of these experiments is to state the significance of disk I/O for out-of-core applications. We can observe that percentage of time spent on I/O (fourth column and seventh column) is more than 61%, with highest for PR application running on WK_FR graph (74%). Therefore, we can conclude that significant amount of time is spent on disk I/O.

The time spent on I/O is significant and out-of-core applications could benefit from two-phased technique. The graph transformation would reduce the input size which would lead to reduction in disk I/O. The straight-forward application of Two-phase technique would require the graph to be reduced significantly in size, however it could not be guaranteed that the disk I/O could be avoided in the first phase. In Section 6.2, we present our modified two-phased technique suited for out-of-core graph processing, which would ensure that the first phase would have no disk I/O.

6.2 Original and Two-Phased Algorithms

We now present our approach for customized Two-phase technique to benefit out-of-core graph application.

Given an iterative vertex-centric graph algorithm $i\mathcal{A}$ and a large input graph $\mathcal{G}$ which does not fit in memory, the accurate results of vertex values $V_{\mathcal{G}}$ can be computed by applying $i\mathcal{A}_o$ (out-of-core execution of $i\mathcal{A}$) to $\mathcal{G}$, that is:

$$V_{\mathcal{G}} \; = \; i\mathcal{A}_o(\mathcal{G})$$

To accelerate out-of-core computation, we use the following steps:

- Partition input $\mathcal{G}$ into n partitions: we partitions the large input graph into n smaller subgraphs $(\mathcal{G}_1, \ldots, \mathcal{G}_n)$ such that each subgraph $\mathcal{G}_i$ fits in memory.
- Reduce each subgraph $\mathcal{G}_i$ to $\mathcal{G}'_i$: we transform each subgraph $\mathcal{G}_i$ into a smaller graph $\mathcal{G}'_i$ via multiple applications of an input reduction transformation $\mathcal{T}$.
- Compute results for $\mathcal{G}'_i$: we apply $i\mathcal{A}_m$ (in-memory execution of $i\mathcal{A}$) to $\mathcal{G}'_i$ to compute $V_{\mathcal{G}'_i}$. Computing on $V_{\mathcal{G}'_i}$ takes place in-memory, hence savings in execution time is achieved.
- Obtain results for $\mathcal{G}$: using simple mapping rules $m\mathcal{R}$s, we convert the results $V_{\mathcal{G}'_i}$ to $V^1_{\mathcal{G}}$. Then, via multiple application of update rules in $i\mathcal{A}_o$, we reduce the error in $V^1_{\mathcal{G}}$ and obtain the result $V^2_{\mathcal{G}}$.

Thus, our approach replaces computation $V_{\mathcal{G}} = i\mathcal{A}_o(\mathcal{G})$ by:

$$
\begin{array}{lll}
\text{[PARTITION GRAPH]} & P(\mathcal{G}) & = [\mathcal{G}_1 \ldots \mathcal{G}_n] \\
\text{[INPUT REDUCTION]} & \mathcal{G}'_i & = \mathcal{T}^{\Delta}(\mathcal{G}_i) \\
\text{[PHASE 1]} & V_{\mathcal{G}'_i} & = i\mathcal{A}_m\ (\mathcal{G}'_i) \\
\text{[MAP RESULTS]} & m\mathcal{R}:\ V_{\mathcal{G}'_i} \rightarrow V^1_{\mathcal{G}} & \\
\text{[PHASE 2]} & V^2_{\mathcal{G}} & = i\mathcal{A}_o\ (V^1_{\mathcal{G}},\ \mathcal{G})
\end{array}
$$

Next, we summarize our approach by presenting the general form of an original out-of-core vertex-centric graph algorithm (Algorithm 8) and its corresponding two-phased version (Algorithm 9). In Algorithm 8, function `IA` represents the original out-of-core algorithm whose

Algorithm 8: Out-of-core Vertex-Centric Graph Algorithm.

```
1  Function iA (input 𝒢)
2      Initialize V_𝒢 & WorkQ
3      while ( ! WorkQ.empty ) do
4          for ( Chunk C : GetGraphFromDisk(𝒢) ) do
5              for ( Vertex v : C ) do
6                  if ( WorkQ.contains(v) ) then
7                      WorkQ.remove(v)
8                      if ( UpdateVals (v, V_𝒢) ) then
9                          WorkQ.add ( outNeighbors (v))
10                     end
11                 end
12             end
13             SaveChunk(C)
14         end
15     end
16     return V_𝒢
17 end
18 Function UpdateVals(v, V_𝒢)
19     Updated ← false
20     if ( updateCheck(v, inNeighbors(v)) ) then
21         update V_𝒢[v]
22         Updated ← true
23     end
24     return Updated
25 end
```

application to graph $\mathcal{G}$ produces the accurate ($V_{\mathcal{G}}$) result. The method GetGraphFromDisk fetches the chunks (`Chunk C`) of graph from disk for processing. The vertex-centric function is applied to the all the vertices in that chunk (lines 4–12). GetGraphFromDisk fetches graph from disk chunk-by-chunk, where each chunk contains a set of vertices and edges and fits in memory. The updated vertex values and edge values are saved back to disk by SaveChunk method call.

In Algorithm 9, function `TPiA` is the two-phased version that calls $\texttt{iA}_m$ (in-memory execution) and $\texttt{iAP}_o$ (out-of-core version) in first and second phases respectively. The

function $\texttt{IA}_m$ represents in-memory execution mode for the original out-of-core algorithm which is used in the first phase and its code sequence is same as the original parallel iterative graph function in Algorithm 6. Note that the processing logic in $\texttt{IA}_o$ (lines 26–38) is exactly same as that in `IA` (lines 3–14). The ΔV stores the vertices that are present in $\mathcal{G}$ but eliminated in the subgraphs $\mathcal{G}'_i$ and, is computed via FindVertexDifference method (line 5) of Algorithm 9. The function $\texttt{IA}_o$ copies results from vertices in $\mathcal{G}'_i$ to vertices in $\mathcal{G}$ for each vertex that is present in both graphs. The vertices in $\mathcal{G}$ that were eliminated in the process of creating $\mathcal{G}'_i$ for each partition $\mathcal{G}_i$ are assigned initial values by `initval()`. Then, similar to `IA`, UpdateVals is applied to $V^2_{\mathcal{G}}$ until convergence. The result ($V^2_{\mathcal{G}}$) is obtained from the application of `TPIA` to $\mathcal{G}$.

6.3 Evaluation

We now evaluate the customized two-phased processing model for the out-of-core graph applications. We implemented baseline and the two-phased version of each of the benchmarks in GraphChi [44], a state-of-the-art out-of-core graph processing system.

Table 6.3 shows the details of the input graphs, their reduced versions, time taken for reduction and the number of partitions to required in the first phase. We use 3 real-world input graphs (Wikipedia-English, Wikipedia-French and LiveJournal) from publicly available Konect repository [42]. To reduce the size of each subgraph we use the tunable parameter **Edge Reduction Percentage (ERP)** as in Equation 4.1. We generate the reduced graphs with varying ERP (75%, 70%, 60%, 50%, 40% and 30%) using our transformation tool based on Algorithm 4.

Algorithm 9: Two Phase Out-of-core Vertex-Centric Graph Algorithm.

```
1  Function TPiA (input 𝒢)
2      [𝒢_1 ... 𝒢_n] ← PartitionGraph (𝒢, n)
3      for ( i → 1 to n ) do
4          𝒢'_i ← ReduceGraph( 𝒢_i, 𝒯, Δ )
5          ΔV ← ΔV + FindVertexDifference ( 𝒢'_i, 𝒢_i )
6          V^1_{𝒢_i} ← iA_m (𝒢'_i)
7      end
8      for ( i → 1 to n ) do
9          V^1_𝒢 ← MergeValues( 𝒢_i, V^1_{𝒢_i} )
10     end
11     V^2_𝒢 ← iAP_o(V^1_𝒢, 𝒢, ΔV)
12     return V^2_𝒢
13 end
14 Function iA_o(V^1_𝒢, 𝒢, ΔV )
15     Initialize WorkQ
16     for ( Vertex v : 𝒢 ) do
17         if ( v ∈ ΔV ) then
18             V^2_𝒢 ( v ) ← inital ( )
19             WorkQ.add ( v )
20         end
21         else
22             V^2_𝒢 ( v ) ← V^1_𝒢 (v)
23         end
24     end
25     WorkQ.add ( Vertex v s.t. v is affected by addition / deletion / partition
        of edges)
26     while (! WorkQ.empty) do
27         for Chunk C : GetGraphFromDisk(𝒢) do
28             for Vertex v : C do
29                 if WorkQ.contains(v) then
30                     WorkQ.remove(v)
31                     if ( UpdateVals (v, V_𝒢) ) then
32                         WorkQ.add ( outNeighbors (v))
33                     end
34                 end
35             end
36             SaveChunk(C)
37         end
38     end
39     return V_𝒢
40 end
```

Table 6.2: Graph Algorithms.

Benchmark	Type
Pagerank (PR) Graph Coloring (GC) Connected Components (CC)	Accurate

Table 6.3: Input Graphs.

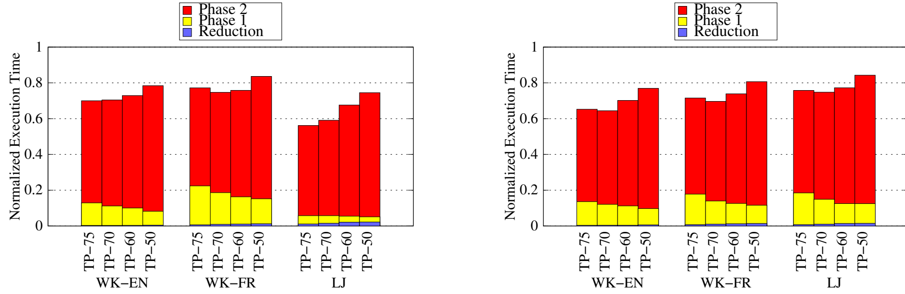
Input Graph		Graph Size		Reduction Time (sec)	#Partitions
		#Nodes	#Edges		
Wikipedia-English (WK-EN)	Original	12.1M	378M	0.83-1.32	15
	Reduced	7.1-9.4M	205-333M		
Wikipedia-French (WK-FR)	Original	3M	102.3M	0.5-0.96	10
	Reduced	1.61-2.5M	65.9-81.5M		
LiveJournal (LJ)	Original	4.8M	69M	0.41-0.83	8
	Reduced	1.2-2.9M	34.1-52.1M		

We performed our experiments on a machine with 4 six-core AMD^{TM} 8431 processors (total 24 cores) and 32 GB RAM running Ubuntu 14.04.1 (kernel version 3.19.0-28-generic). The programs were compiled using GCC 4.8.4, optimization level `-O3`.

We evaluate the performance of following versions of the benchmark implementations:

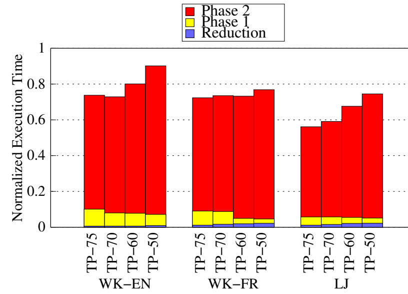
- **Baseline:** based on the traditional processing model.
- **TP-X:** based on our two-phased processing model using reduced graphs with ERP = X%. Note that the execution times include the graph reduction times which are already presented in Table 6.3.

Efficiency of Two-Phased Processing. Figure 6.2 shows the speedups achieved by TP-X over Baseline for X $\in \{50\%, 60\%, 70\%, 75\%\}$. On an average, ERP 70% has the



(a) Normalized Execution times for PR. For comparison, the Baseline execution times (in sec) for WK-EN, WK-FR and LJ are 272.75, 75.88 and 61.77 respectively.

(b) Normalized Execution times for GC. For comparison, the Baseline execution times (in sec) for WK-EN, WK-FR and LJ are 206.76, 67.65 and 54.94 respectively.



(c) Normalized Execution times for CC. For comparison, the Baseline execution times (in sec) for WK-EN, WK-FR and LJ are 140.81, 43.5 and 36.65 respectively.

Figure 6.2: Normalized execution time of two-phased execution for each benchmark-graph-ERP value.

maximum speedup (1.48 $\times$). We can observe that the speedups decrease as ERP decreases from 70% to 50%; on an average TP-70, TP-60 and TP-50 achieve a speedup of 1.48$\times$, 1.38$\times$ and 1.25$\times$ respectively. This because the savings achieved in phase 1 for TP-70, TP-60 and TP-50 is minimal w.r.t. to the additional time taken to reduce the error in phase 2 which uses out-of-core execution. On average for TP-70, TP-60 and TP-50, the savings achieved in the second phase are 44.15%, 42.45%, 36.76% and 38.54% respectively.

The average speedup increase as ERP decreases from 75% (1.47$\times$) to 70%(1.48 $\times$). This is because the savings is achieved in phase 1 subsumes the additional time required in

phase 2 for 70% ERP, since the 70% ERP subgraphs are structurally less similar to original graph than 75% ERP.

The savings achieved in phase 1 w.r.t. *Baseline* for TP-75, TP-70, TP-60 and TP-50 are 87.97%, 90.09%, 91.71% and 92.65%. These high savings achieved in the allow tolerating the execution times of reduction and second phase over original graphs; the execution times normalized w.r.t. *Baseline* for the second phase of TP-75, TP-70, TP-60 and TP-50 are 0.56, 0.58, 0.63, and 0.71 respectively and for the reduction are as low as 0.01 for all the ERP. Since our reduction transformations are local and non-interfering, the cost of performing the input reduction is much lower than the savings achieved in processing the graph in out-of-core fashion.

As expected, the time taken to process the reduced graph in the first phase decreases as ERP decreases simply because the work done is typically proportional to the size of subgraph. On the other hand, the execution time in second phase increases as ERP decreases. This is mainly because an aggressively reduced graph with lower ERP is structurally less similar to the original graph compared to that reduced with a higher ERP. Hence, the values which are fed from reduced graph with lower ERP require more computation in the second phase in order to reach to maximum possible accuracy for the original graphs.

To visualize the reduction in I/O caused by our technique, we measure the time spent in I/O for both Baseline and TP-X execution. The average I/O savings achieved using TP-X over Baseline for X $\in \{75\%, 70\%, 60\%, 50\%\}$ are 1.92, 1.86, 1.71 and 1.50. The savings decreases with decrease in ERP value, because the amount of work to be done in the second phase increases as ERP decreases. Figure 6.3 shows savings in I/O for PR, CC and GC

using ERP-70 (most efficient ERP setting). The savings increases as size of graph increases with WK-EN having the largest amount of savings. The reason for such behavior is the I/O savings achieved by in-memory execution of first phase is significant larger graphs as compared to the smaller graphs.

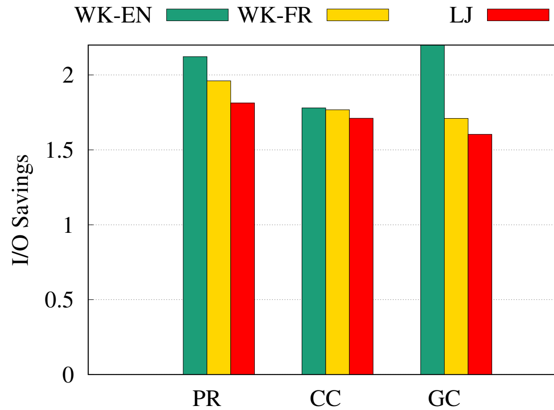


Figure 6.3: Savings in I/O using ERP-70 for PR, GC & CC.

Slowdown for Path-Based Applications. We ran experiments on other path-based applications – SSSP and SSWP. Our proposed technique could not accelerate these applications because of the following reason. We use edge-cut partitioning technique to partition the input graph, which distributed vertices across multiple partitions. It is not guaranteed that each vertex in the input graph will be present in every partition. SSSP and SSWP computes the shortest and widest path respectively from a source vertex. The partitioning scheme does not ensure the source vertex to be present in every partitions. There will be partitions where the source vertex will not be present, all the vertex values will remain in their initialized states. The results obtained from the first phase are not useful for many partitions, which increases the amount of work to be done in the second phase.

Overhead of Two-Phased Method. Our customized Two Phased method creates multiple partitions for the input graph such that the first phase could be carried out in-memory. The partitions are reduced and processed separately, therefore additional disk space is required. Our approach imposes a disk overhead of $\approx 2\times$ to store these additional subgraphs.

6.4 Out-of-Core Graph Partitioning

To ensure the first phase is carried out in-memory, there is a need to partition the graph into smaller subgraphs before each subgraph is transformed. Metis [34] has provided an in-memory partitioning scheme which requires graph to fit in memory. The partitioning technique in PowerGraph [24] doesnot require the entire graph to load in-memory. However, this partitioning scheme introduces more edge-cuts unlike Metis. With increase in edge-cuts in partitioning, the phase 1 generate lesser useful approximation of result, which makes the phase 2 (the I/O costly phase) work more. Hence, we need an out-of-core partitioning technique which introduces less edge-cuts like Metis [34].

We now present our out-of-core partitioning algorithm which is based on multilevel k-way partitioning scheme by Karypis [34]. Given a large input graph $\mathcal{G}$ which does not fit in memory, and number of partitions *nparts*, our out-of-core algorithm would make partitions with minimal edge-cuts. Partitioning has 3 major steps: the coarsening step (lines 4–14), the initial partitioning step (lines 15–20) and the uncoarsening step (lines 21–24). Coarsening step follows recursion where in each step, graph $\mathcal{G}_{i+1}$ is created from $\mathcal{G}_i$ by randomly matching vertices of the graph $\mathcal{G}_i$. Since the graph doesnot fit in memory, the

chunks of graphs are fetched from disk via FetchGraphFromDisk function and each vertex is matched with its neighbors using the method RandomMatchVertex. Therefore, in each step of recursion $\mathcal{G}_{i+1}$ is has lesser number of vertices than $\mathcal{G}_i$. The recursion is terminated when either of two conditions are met: 1) The number of vertices in $\mathcal{G}_i = nparts$. 2) No further vertex pair can be matched. Initial partitioning step is done on $\mathcal{G}_i$ graph, where each vertex is assigned to the partition which has the least number of vertices and, the partition data is saved in PD_i.

The uncoarsening phase is done iteratively by refining the partition for each coarser graph $\mathcal{G}_j$ till $\mathcal{G}_0$ which is same as $\mathcal{G}$ (line 2). In each iteration the following steps are done: 1) The partition data PD_j is loaded for graph $\mathcal{G}_j$. 2) RefinePartition function is called which projects the partition PD_j on $\mathcal{G}_{j-1}$ and generates PD_{j-1} by refining the partition for $\mathcal{G}_{j-1}$. The partition refinement step incrementally swaps vertices among the partitions to reduce the number of edge-cut in the partitioning until a local minima is reached. This final step ensures the number of edge-cut to be minimal for partitioning.

6.5 Summary

In this chapter we showed that time spent on I/O for the out-of-core graph applications is significant (up to 74%). Therefore, our two-phased processing model could benefit out-of-core applications by reducing I/O. We then presented a modified version of two-phased technique, which reduces the amount of I/O in out-of-core graph application. The first phase is carried out in-memory, however the second phase is still executed in out-of-core fashion. The processing logic of phase 1, phase 2, and original graph algorithms

Algorithm 10: Out-of-core Graph Partitioning Algorithm.

```
1  Function PartitionGraph (𝒢, nparts)
2      i ← 0
3      𝒢_i ← 𝒢
4      while ( !(VertexMatchLimitReached(𝒢_i)) && nVertices(𝒢_i) < nparts )
        do
5          𝒢_{i+1} ← initialize
6          for ( Chunk C:FetchGraphFromDisk(𝒢_i) ) do
7              for ( Vertex v: C ) do
8                  RandomMatchVertex(v, neighbors(v))
9              end
10             SaveChunk(C, 𝒢_{i+1})
11         end
12         i ← i + 1
13         𝒢_i ← 𝒢_{i+1}
14     end
15     for ( Chunk C: FetchGraphFromDisk(𝒢_i) ) do
16         PD_i ← initialize
17         for ( Vertex v: C ) do
18             PD_i.add(v, nparts) // s.t. each vertex gets assigned to partition
                with least number of vertices)
19         end
20     end
21     for ( j ← i to 1 ) do
22         PD_j ← LoadPartitionData (𝒢_j)
23         RefinePartition(𝒢_{j−1}, PD_j)
24     end
25     return PD_0
26 end
```

is similar, except phase 1 store the entire graph in memory. Our evaluation shows that average execution savings upto 1.4× could be achieved and I/O could be saved by 1.75×. Finally, we developed an out-of-core partitioning technique which introduces fewer edge-cuts than Metis [34].

Chapter 7

Related Work

This chapter summarizes various prior research in domains and problems addressed by this thesis. We first summarize the existing solutions for data structure selection and runtime adaptation techniques. Next, we present the related work for data transformation and approximate computing. Finally, we summarize the work for out-of-core graph processing and partitioning techniques.

7.1 Data Structure Selection and Runtime Adaptation.

Data Structure Selection. There has been a large body of work that helps the programmer to select the data structure during the runtime. Jung et al. developed DDT [29], a dynamic program analysis tool that identifies the data structures used by executing the application binary with the objective of identifying problems in the data structure choice with respect to a particular compiler and microarchitecture. Their followup work developed the Brainy [30] tool which predicts the best data structure for a particular program

input and underlying architecture. They rely on the cost model of the data structure for a particular input and architecture combination. Sharir et al. proposed data structure selection techniques for SETL [70] programs, where the programmer codes the algorithms while the data structures are selected automatically at compile-time based on instructions in the program. Perflint [45], a tool for identification of suboptimal patterns in C++, suggests data structure changes to the programmer. Chameleon [72] assists the programmer in the choice of collections in Java and C#. These approaches are offline, hence do not support adaptation when input characteristics change during a single run.

Huang et al. proposed Self Adaptive Containers [27] where they provide the developer with a container library which adjusts the underlying data structure associated with the container to meet Service Level Objectives (SLO) specified by the developer. During runtime, the adaptation of underlying data structure in the container takes place whenever there is a violation in the specified SLO. Similarly, Xu proposed a technique named CoCo [89] for runtime replacement of Java collections, depending on a predefined condition e.g., container size. The developer must add the abstractions and concretization operations for the container classes manually. CoCo generates glue code to join the container classes and make a combo object, which is capable of profiling and container replacement. Our approach differs from these two on several dimensions. We deliberately avoid manual specification of abstraction and concretion operations or SLO for the underlying data structure, because such identification is tedious, error-prone and discourages programmers from using off-the-shelf code. Self Adaptive Containers and CoCo replace some standard collections, which have a standard but narrow interface, and no scope for user-defined data structures. We allow

general replacement of any user-defined data structure and their methods without any constraint on the type of operations. CoCo does not consider the space-time trade-off. While Self Adaptive Containers specifications may contain both response time and memory consumption for a particular service, those are fixed for the entire course of execution rather than catering to the need for change in input/workload characteristics.

Runtime Adaptation. One way to effect adaptation is via programming support. In our approach we consider the data structure to be the key for the adaptation. Our infrastructure offers programming support for specifying adaptation policies, as well as alternative versions of data structures and implementations among which execution can be switched to match the current operating conditions. We target applications written in C. Ghezzi et al. have developed ContextErlang [23], which implements Context-Oriented Programming in Erlang and support the construction of self-adaptive software. They implement context dependent behavior by allowing messages to be processed by variations (different callback modules). The dynamic binding of these variations are available in Erlang. The Elastin framework[54] allows runtime adaptation via different configuration. These different configurations are combined into a single elastic application which can switch the configurations on the fly at runtime. This approach does not take into consideration of the changes in the input data property as a trigger for the adaptation. Their adaptation is done via a dynamic software update tool Ginseng [55, 58], which allows the adaptation at safe update points. This adaptation does not occur at the middle of a computation and it has no measure to adapt the saved result of a computation. Its configuration does not take into consideration the change in the input data characteristics.

7.2 Data Transformations and Approximate Computing

Graph processing has gained a lot of attention due to its applicability across various domains. Many graph processing frameworks have been developed for distributed ([48, 47, 84, 88, 68]), shared memory ([62, 74, 85]) and GPU-based environments ([37, 71]). These frameworks include a parallel runtime that iteratively processes the input graph until all the graph values converge. The computation is based on asynchronous or bulk synchronous model [82]. This traditional style of processing includes a single processing phase.

Multilevel Transformation Techniques. There is a body of work [26, 52, 7, 53, 67, 91, 35, 33, 36, 86] that reduces the size of graphs to accelerate processing. These works mainly rely on algorithm-specific reduction techniques and mostly operate of regular meshes. [26] presents a multilevel graph partitioning algorithm where first a hierarchy of smaller graphs is created, then the highest level graphs are partitioned and then, these partition results are carefully propagated back down the hierarchy to achieve partitioning of the original graph. It uses edge contraction where neighbors are unified into a single vertex which is suitable for relatively regular meshes. [52] uses the same three phases and relies on quality functions of the reduced (coarse) grids based on aspect ratio. Moreover, the reduction algorithm operates on the dual graph and uses maximal independent set computation which requires non-trivial processing. [7, 35, 33, 36] also aim to partition graphs via recursive edge contraction using maximal independent set computation and edge contraction to generate multinodes. In contrast, our work identifies light-weight, local and non-interfering transformations that are *general* (i.e., not algorithm-specific) and effective for *irregular* input graphs. Moreover, our

reduction strategy is not hierarchical (multi-level) since our transformations are designed from the vertex's perspective and are applied at most once on each vertex. [86] processes queries by providing multiple levels of abstractions and refining the query to these abstraction levels. [67] is specifically designed for distance based algorithms like SSSP where they aim to achieve gate vertex sets which allow traversals to be constructed on the reduced graphs. [91] reduces by pruning weakest edges based on cost functions and which adhere to specific constraints related to connectivity maintenance. These works require path- or component-level transformations that are computationally expensive whereas our transformations are light-weight and hence effective for large graphs.

Beyond these works, various optimization techniques have been developed which attempt to accelerate processing at the cost of achieving approximate results. We divide the literature encompassing such approximation based graph processing techniques into two categories, discussed below. None of these techniques provide correctness guarantees and hence the results of these techniques are always approximate.

Algorithm-specific Approximation. Chazelle et al. proposed a technique for approximating the weight of minimum spanning tree in sublinear time by approximating the number of connected components [14]. The technique approximates the weight of the minimum spanning tree but it does not find the tree. Nanongkai [53] proposed an approximation technique to find SSSP and all-pair shortest path (APSP) by bounding the diameter of the graph. Bader et al. [4] proposed the approximation of Betweenness Centrality (BC) by employing an adaptive sampling technique. The algorithm samples a subset of vertices and performs SSSP on them selected, thus reducing the number of SSSP operations to determine

BC. In contrast to sampling, our approach to input graph reduction is smarter as it considers graph connectivity and is more general as it is applied to a class of graph algorithms. In fact all of the above approximation techniques were developed for a single specific graph application. In contrast, *our technique applies to many iterative graph algorithms all using the same input reduction transformations.*

Compiler-based Approximation. Researchers have focused on trading accuracy for execution time by skipping a task's execution or by choosing a specific implementation from multiple ones provided by the developer. Rinard proposed early termination [65] and task skipping [64] that are applied during execution. These techniques use a distortion model based on sampling to estimate the error introduced due to early termination or task skipping. The work does not provide an empirical justification for the distortion model and thus it is unclear if it will work for input graphs with different characteristics. Green [5] selects a specific implementation out of many different implementations provided by the developer while maintaining the quality of output. Hoffman et al. [75] proposed loop perforation where certain iterations of a loop are skipped to trade off accuracy for faster execution. The loops that are perforated are chosen with the help of training input and the error bound set by the user. This technique is not useful for different graph applications since it requires perforated loops to fall into one of the specified categories of the global patterns. *Our technique does not require loops to follow any such pattern and it does not perform any static or dynamic analysis of the application to achieve approximation.*

The Sage [69] compiler generates CUDA kernels that exploit GPUs to achieve approximation using different optimizations. The runtime system includes a tuning phase

which selects the best optimization technique and a calibration phase to help maintain quality. Although we tested our methodology only for CPU systems, it can be easily applied for GPUs as we achieve approximation by reducing the input graph. Shang et al. [73] proposed auto-approximation of vertex-centric graph applications by automatically synthesizing the approximate version of an application. They combined different approximation techniques such as task skipping, sampling, memorization, interpolation and system function replacement for synthesizing the approximate version. Carbin et al. [12] proposed a language to specify approximate program transformations. *Our approach works without modifying the original implementation. Moreover input reductions, guided by impact on graph connectivity, customize the skipped computations to input characteristics.*

7.3 Out-of-core Graph Processing

Researchers have focussed on improving out-of-core graph processing via different data organization techniques. Infinimem [40] has introduced language and runtime support for size-oblivious programming, where a program is written without the concern of input size, i.e., if the data does't fit in the memory then it will be stored in disk as object. Similarly, Graphchi [44] uses `SHARDS` (a sorted collection of edges) in the disk for storing the input graph. *Our technique doesn't rely on any specific representation in disk and it could be easily applied to any out-of-core graph processing system which uses vertex-centric programming model.* X-Stream [66] uses *streaming partitions* (a vertex set, an edgelist and an updatelist) for storing graph in disks, however they require the programs to be written in edge-centric fashion. We developed a vertex-centric system as it is easy for the programmer.

Chapter 8

Conclusions and Future Directions

8.1 Contributions

This dissertation contributes to enhancing the *performance* and *functionality* of Big Data applications. The contributions are divided into three parts: 1) a runtime technique for switching between the representation of data in memory to match the input/workload characteristics; 2) data transformation techniques for optimized processing of data via a two-phased processing model; and 3) extending the two-phased processing model to handle out-of-core applications which use disk as an extension to memory. Next, we highlight the novel aspects of each of these above contributions.

8.1.1 Alternate Data Structure Representations

The performance of Big Data applications is sensitive to the choice of data structure used to hold the data being processed. This dissertation investigates an important aspect of data structure selection. For applications where the input/workload characteristics can

change over time, the best performing data structure choice also changes with time. Therefore, choice of data structure must be *adapted* based upon *input/workload characteristics*. This dissertation presents a runtime technique for switching the data structure *dynamically* and *safely*, when there is a mismatch between the data structure choice and *input/workload characteristics*. This technique addresses multiple challenges: first, it is not difficult for the programmer to use; second, the switching between the alternative representation is safe; third, the system responds quickly to changes in the input/workload characteristics.

The above challenges are addressed as follows. The programmer is provided with a small set of annotations whose use requires minor changes in the source code. The static analysis and the compilation tool takes care of the identification of program points for *safely* and *timely* switch of the data structure representation. The experiments presented in the dissertation demonstrate that runtime technique allows applications to react quickly to the input/workload characteristics changes, with little execution time and memory overhead.

8.1.2 Data Transformations

The sizes of data processed by Big Data applications are (unsurprisingly) large, thus so is the memory footprint. The data held in memory is accessed multiple times during execution, which leads to significant amount of time being spent on data access. This dissertation provides a processing model called two-phased processing model, where the input is transformed to reduce the memory footprint and thus reducing the data access. This transformed input is used for majority of the computation and thus accelerates the execution by cutting down the time spent on data access.

The two-phased processing model presented in this dissertation is primarily focused

on accelerating vertex-centric graph applications. The most important aspect of this processing model is its *input-data centric* data transformation and requires no change in original vertex centric algorithm. In the first phase, the original (unchanged) iterative algorithm is applied on a transformed input (smaller graph) which is representative of the original large input; this step yields savings in execution time. In the second phase, the results from the smaller graph are transferred to the original larger graph and, via application of the original graph algorithm, error reduction is achieved, possibly converging to the final accurate results. This acceleration could be applied on multiple processing environment such as GPU, Distributed System and out-of-core Systems.

8.1.3 Out-of-core Computations

Often the size of input for Big Data applications is so large that it cannot be held in memory. The programmers have introduced the out-of-core programming model to handle such scenarios, where a persistent storage is used as an extension to memory. The data is brought into memory for processing, and after the processing is done, data is stored back into disk. This leads to a significant amount of time being spent on the disk I/O.

This dissertation presents a technique which scales the *functionality* of the graph application by extending the two-phased execution to reduce I/O. It uses the edge-cut reducing partitioning algorithm to partition the input graph, so that each partitioned subgraph could be transformed in-memory to reduce the input size; this step saves the time spent on disk I/O. In the second phase, the results from these subgraphs are transferred to original graph and, final processing is done in regular out-of-core fashion. This technique

requires no change in original out-of-core implementation and, efficiently uses *input-data centric* data transformation to reduce disk I/O.

8.2 Future Work

Data Storage Optimization. Data storage has become one of the most important application of cloud computing, where the users can remotely store their data into cloud and, often the cloud provides persistent data storage devices. The data storage system manages and backs up the data remotely, and the data representation for reading and writing secondary storage can greatly impact application performance. The alternate data structure representation technique enhances functionality and optimizes the disk usage for these data storage system. Therefore it is an important future direction to use alternate data structure representation for optimizing data storage for cloud computing.

Application to Diverse Domains. The multiple data representations technique could be applied to other domains such as bioinformatics and computational genomics where the input representations matters. They process graph-like connected structure which could greatly benefit from the multiple data representation technique. Therefore, future work can explore applicability and customization of our solutions to these domains.

This dissertation has presented the alternate data structure technique corresponding to the changes in input/workload characteristics. The alternate data structure representations technique could be extended to save energy by switching to more energy efficient data structure during execution. Future work in this direction requires detailed study of energy profile of data structure representation.

Our safe switching technique between alternate data structure representation could be further extended to enhance the data structure obfuscation, by switching multiple times between obfuscated alternate data structure. This would serve as an important future work, as exploring this direction would produce enhanced security measures and increase the difficulty level of reverse engineering.

Scaling to Large Clusters. The experiments done in this dissertation are limited to multi-core systems. Scaling the multiple data representation techniques to distributed system could expose multiple challenges which were not encountered on a single machine (e.g., the overhead of communication cost and the data representation synchronization across multiple nodes).

Bibliography

[1] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49, 2009.

[2] J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis. Enabling autonomic behavior in systems software with hot swapping. *IBM Syst. J.*, 42(1):60–76, January 2003.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, 2012.

[4] David A. Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. Approximating betweenness centrality. In *Proceedings of the 5th International Conference on Algorithms and Models for the Web-graph*, WAW'07, pages 124–137, Berlin, Heidelberg, 2007. Springer-Verlag.

[5] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM.

[6] Sumita Barahmand and Shahram Ghandeharizadeh. Bg: A benchmark to evaluate interactive social networking actions. In *CIDR'13*.

[7] Stephen T. Barnard and Horst D. Simon. Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience*, 6(2):101–117, 1994.

[8] Major Bhadauria and Sally A. McKee. An approach to resource-aware co-scheduling for cmps. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 189–199, 2010.

[9] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, 2011.

[10] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. Storage strategies for collections in dynamically typed languages. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 167–182, 2013.

[11] Angela Demke Brown and Todd C. Mowry. Taming the memory hogs: Using compiler-inserted releases to manage physical memory intelligently. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 3–3, 2000.

[12] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 169–180, New York, NY, USA, 2012. ACM.

[13] F. Chang and V. Karamcheti. Automatic configuration and run-time adaptation of distributed applications. In *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on*, pages 11 –20, 2000.

[14] Bernard Chazelle, Ronitt Rubinfeld, and Luca Trevisan. Approximating the minimum spanning tree weight in sublinear time. *SIAM J. Comput.*, 34(6):1370–1379, June 2005.

[15] Haibo Chen, Jie Yu, Chengqun Hang, Binyu Zang, and Pen-Chung Yew. Dynamic software updating using a relaxed consistency model. *Software Engineering, IEEE Transactions on*, 37(5):679–694, Sept 2011.

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, 2010.

[17] Timothy A. Davis. *The University of Florida Sparse Matrix Collection*, volume 92. 1994.

[18] Brian Demsky and Martin Rinard. Automatic data structure repair for self-healing systems. In *In Proceedings of the 1 st Workshop on Algorithms and Architectures for Self-Managing Systems*, 2003.

[19] I. S. Duff, Roger G. Grimes, and John G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Softw.*, 15(1):1–14, March 1989.

[20] Bin Fan, David G. Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 371–384, 2013.

[21] Ayman Farahat, Thomas LoFaro, Joel C. Miller, Gregory Rae, and Lesley A. Ward. Authority rankings from HITS, pagerank, and SALSA: Existence, uniqueness, and effect of initialization. *SIAM J. Sci. Comput.*, 27(4):1181–1201, November 2005.

[22] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. *Rainbow: architecture-based self-adaptation with reusable infrastructure*, volume 37. oct. 2004.

[23] Carlo Ghezzi, Matteo Pradella, and Guido Salvaneschi. Programming language support to context-aware adaptation: a case-study with erlang. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 59–68, 2010.

[24] 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, Hollywood, CA, 2012. USENIX.

[25] R. Gupta and R. Bodik. Adaptive loop transformations for scientific programs. In *Parallel and Distributed Processing, 1995. Proceedings. Seventh IEEE Symposium on*, pages 368–375, Oct 1995.

[26] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[27] Wei-Chih Huang and William J. Knottenbelt. Self-adaptive containers: Building resource-efficient applications with low programmer overhead. In *SEAMS*, pages 123–132, 2013.

[28] James Kobielus. Graph analysis will make big data even bigger, July 2013. `http://www.infoworld.com/d/big-data/graph-analysis-will-make-big-data-even-bigger-223420`.

[29] Changhee Jung and N. Clark. Ddt: Design and evaluation of a dynamic program analysis for optimizing data structure usage. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 56–66, Dec 2009.

[30] Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. Brainy: Effective selection of data structures. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 86–97, 2011.

[31] Jure Leskovec. Stanford Network Analysis Platform. `http://snap.stanford.edu/snap/index.html`.

[32] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. Software data spreading: Leveraging distributed caches to improve single thread performance. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 460–470, 2010.

[33] George Karypis and Vipin Kumar. Multilevel graph partitioning schemes. In *Proc. 24th Intern. Conf. Par. Proc., III*, pages 113–122. CRC Press, 1995.

[34] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1), 1998.

[35] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[36] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, January 1998.

[37] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Scalable SIMD-efficient graph processing on gpus. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT '15, pages 39–50, 2015.

[38] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. CuSha: Vertex-centric graph processing on gpus. HPDC '14, pages 239–252.

[39] R. Knauerhase, P. Brett, B. Hohlt, Tong Li, and S. Hahn. Using os observations to improve performance in multicore systems. *Micro, IEEE*, 28(3):54–66, May 2008.

[40] Sai Charan Koduru, Rajiv Gupta, and Iulian Neamtiu. Size oblivious programming with infinimem. LCPC'15.

[41] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. Adapt: A framework for coscheduling multithreaded programs. *ACM Trans. Archit. Code Optim.*, 9(4):45:1–45:24, 2013.

[42] Jérôme Kunegis. KONECT: The koblenz network collection. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13 Companion, pages 1343–1350, New York, NY, USA, 2013. ACM.

[43] Amlan Kusum, Iulian Neamtiu, and Rajiv Gupta. Adapting graph application performance via alternate data structure representation. In *ADAPT'15*.

[44] 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.

[45] Lixia Liu and S. Rus. Perflint: A context sensitive performance advisor for c++ programs. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, pages 265–274, 2009.

[46] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. GraphLab: A new framework for parallel machine learning. *ArXiv e-prints*, August 2014.

[47] 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. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[48] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[49] J. Mars and R. Hundt. Scenario based optimization: A framework for statically enabling online optimizations. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, pages 169–179, March 2009.

[50] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 248–259, 2011.

[51] Memcached. `http://memcached.org/`.

[52] Irene Moulitsas and George Karypis. Multilevel algorithms for generating coarse grids for multigrid methods. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 45–45, New York, NY, USA, 2001. ACM.

[53] Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, STOC '14, pages 565–573, New York, NY, USA, 2014. ACM.

[54] Iulian Neamtiu. Elastic executions from inelastic programs. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 178–183, 2011.

[55] Iulian Neamtiu and Michael Hicks. Safe and timely updates to multi-threaded programs. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 13–24, 2009.

[56] Iulian Neamtiu and Michael Hicks. Safe and timely updates to multi-threaded programs. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 13–24, 2009.

[57] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 37–49, 2008.

[58] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for c. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 72–83, 2006.

[59] P. Oreizy, N. Medvidovic, and R.N. Taylor. Architecture-based runtime software evolution. In *Software Engineering, 1998. Proceedings of the 1998 International Conference on*, pages 177 –186, apr 1998.

[60] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime software adaptation: Framework, approaches, and styles. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 899–910, 2008.

[61] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal Of Algorithms*, 51(2), 2004.

[62] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[63] K.K. Pusukuri, R. Gupta, and L.N. Bhuyan. No more backstabbing... a faithful scheduling policy for multithreaded programs. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 12–21, Oct 2011.

[64] Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 324–334, New York, NY, USA, 2006. ACM.

[65] Martin C. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 369–386, New York, NY, USA, 2007. ACM.

[66] 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.

[67] Ning Ruan, Ruoming Jin, and Yan Huang. Distance preserving graph simplification. In *Proceedings of the 2011 IEEE 11th International Conference on Data Mining*, ICDM '11, pages 1200–1205, Washington, DC, USA, 2011. IEEE Computer Society.

[68] Semih Salihoglu and Jennifer Widom. GPS: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 22:1–22:12, New York, NY, USA, 2013. ACM.

[69] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. SAGE: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 13–24, New York, NY, USA, 2013. ACM.

[70] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. An automatic technique for selection of data representations in setl programs. *ACM Trans. Program. Lang. Syst.*, 3(2):126–143, 1981.

[71] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. GraphReduce: Processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 28:1–28:12, New York, NY, USA, 2015. ACM.

[72] Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: Adaptive selection of collections. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 408–418, 2009.

[73] Zechao Shang and Jeffrey Xu Yu. Auto-approximation of graph computing. *Proc. VLDB Endow.*, 7(14):1833–1844, October 2014.

[74] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[75] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 124–134, New York, NY, USA, 2011. ACM.

[76] Craig A. N. Soules, Jonathan Appavoo, Dilma Da Silva, Marc Auslander, Gregory R. Ganger, Michal Ostrowski, and et al. System support for online reconfiguration, 2003.

[77] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, 1996.

[78] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 29(4), August 2007.

[79] Lingjia Tang, Jason Mars, and Mary Lou Soffa. Compiling for niceness: Mitigating contention for qos in warehouse scale computers. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 1–12, 2012.

[80] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 283–294, 2011.

[81] Lingjia Tang, Jason Mars, Wei Wang, Tanima Dey, and Mary Lou Soffa. Reqos: Reactive static/dynamic compilation for qos in warehouse scale computers. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS '13, pages 89–100, 2013.

[82] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[83] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2Nd ACM Workshop on Online Social Networks*, WOSN '09, pages 37–42, 2009.

[84] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. ASPIRE: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 861–878, New York, NY, USA, 2014. ACM.

[85] Guozhang Wang, Wenlei Xie, Alan J Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[86] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. GraphQ: Graph query processing with abstraction refinement—scalable and programmable analytics over very large graphs on a single pc. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 387–401, Santa Clara, CA, July 2015. USENIX Association.

[87] Wei Wang, T. Dey, J. Mars, Lingjia Tang, J.W. Davidson, and M.L. Soffa. Performance analysis of thread mappings with a holistic view of the hardware resources. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 156–167, April 2012.

[88] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. GraphX: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

[89] Guoqing Xu. Coco: Sound and adaptive replacement of java collections. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 1–26, 2013.

[90] Youtao Zhang and Rajiv Gupta. Data compression transformations for dynamically allocated data structures. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 14–28, 2002.

[91] Fang Zhou, Sebastien Malher, and Hannu Toivonen. Network simplification with minimal loss of connectivity. In *Proceedings of the 2010 IEEE International Conference on Data Mining*, ICDM '10, pages 659–668, Washington, DC, USA, 2010. IEEE Computer Society.

[92] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 129–142.