ANALYSIS OF PARALLEL ITERATIVE GRAPH APPLICATIONS ON SHARED MEMORY SYSTEMS

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

By

Funda Atik

January 2018

ANALYSIS OF PARALLEL ITERATIVE GRAPH APPLICATIONS
ON SHARED MEMORY SYSTEMS
By Funda Atik
January 2018

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Özcan Öztürk(Advisor)

Uğur Güdükbay

Süleyman Tosun

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

ANALYSIS OF PARALLEL ITERATIVE GRAPH APPLICATIONS ON SHARED MEMORY SYSTEMS

Funda Atik
M.S. in Computer Engineering
Advisor: Özcan Öztürk
January 2018

Graph analytics have come to prominence due to their wide applicability to many phenomena of real world such as social networks, protein-protein interactions, power grids, transportation networks, and other domains.

Despite the increase in computational capability of current systems, developing an effective graph algorithm is challenging due to the complexity and diversity of graphs. In order to process large graphs, there exist many frameworks adopting different design decisions. Nonetheless, there is no clear consensus among the frameworks on optimum design selections.

In this dissertation, we provide various parallel implementations of three representative iterative graph algorithms: Pagerank, Single-Source Shortest Path, and Breadth-First Search by considering different design decisions such as the order of computations, data access pattern, and work activation. We experimentally study the trade-offs between performance, scalability, work efficiency of each implementation on both real-world and synthetic graphs in order to guide developers in making effective choices while implementing graph applications.

Since graphs with billions of edges can fit in memory capacities of modern shared-memory systems, the applications are implemented on a shared-memory parallel/multicore machine. We also investigate the bottlenecks of each algorithm that may limit the performance of shared-memory platforms by considering the micro-architectural parameters.

Finally, we give a detailed road-map for choosing design points for efficient graph processing.

Keywords: Shared Memory, Graph Applications, Parallel Programming.

ÖZET

ORTAK BELLEKLİ SİSTEMLER ÜZERİNDE ÇALIŞAN PARALEL TEKRARLAYAN ÇİZGE UYGULAMALARININ ANALİZİ

Funda Atik
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Özcan Öztürk
Ocak 2018

Çizge analiz uygulamaları; sosyal ağlar, proteinler arası etkileşimler, güç nakli şebekeleri, taşıma ağları gibi birçok alana uygulanabilirlikleri sayesinde gittikçe önem kazanmıştır.

Çizge veri setlerinin karmaşık ve çok çeşitli yapıda olması nedeniyle, günümüz sistemlerinin sayısal işlem yapabilme kapasitesi artsa da etkili bir çizge algoritması geliştirmek son derece zordur. Büyük çizge veri setlerini işlemek amacıyla önceden hazırlanmış yapıların ve fonksiyonların bulunduğu farklı dizayn kararları alan birçok çerçeveler geliştirilmiştir. Fakat bu çerçeveler arasında en iyi tasarım seçimlerinin nasıl olacağı hakkında ortak bir karar yoktur.

Bu tezde; işlemlerin uygulanma sırası, veri erişim biçimleri ve iş aktivasyonu gibi farklı tasarım kararları göz önüne alınarak, tekrarlayan çizge uygulamalarına örnek teşkil eden üç algoritmanın çeşitli paralel geliştirmeleri sunulmuştur. Bu üç algoritma şunlardır: "PageRank", "Tek Kaynaklı En Kısa Yollar" ve "Sığ Öncelikli Arama". Her bir uygulamanın, hem gerçek hem de sentetik çizge veri setleri üzerinde performans, ölçeklenebilirlik ve iş verimi analizleri yapılarak bunlar arasında nasıl bir denge kurulabileceği deneysel olarak araştırılmıştır.

Milyarlarca düğüm ve bunları birbirine bağlayan çok sayıda bağıntıdan oluşan çizge veri setleri her ne kadar çok büyük olsalar da modern ortak bellekli sistemlerin bellek kapasitesine sığabilirler. Bu nedenle, bu tezde, tüm uygulamalar ortak bellekli paralel çok çekirdekli sistemler üzerinde tasarlanmıştır. Aynı zamanda donanım performans sayaçları kullanılarak ortak bellekli sistemlerin performansını sınırlayabilecek noktaları belirlemek için her bir algoritmanın mikro-donanımsal değişkenleri incelenmiştir.

Sonuç olarak; bu tezde, geliştiricilerin çizge analiz uygulamaları yazarken, farklı tasarım kararları arasından etkili ve bilinçli seçimler yapabilmeleri hedeflenmiştir.

Anahtar sözcükler: Ortak Bellek, Çizge Uygulamaları, Paralel Programlama.

Acknowledgement

First, I would like to thank my advisor Özcan Öztürk for his support during my M.S studies. He provided me with invaluable lessons.

I am also grateful to members of my thesis committee, Uğur Güdükbay and Süleyman Tosun, for their interest on this topic.

I owe my deepest gratitude to Ali Can Atik, Şerif Yeşil, and Fulya Atik for their continual encouragement, especially when it was most needed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the advent of big data applications, graph analytics has gained much importance in recent years. Applications of graph analytics could be applied to a wide range of domains such as biology, robotics, machine learning, and artificial intelligence [1]. Moreover, a huge amount of data can be represented as graphs such as biological networks, social networks, road networks, item-product networks, and so on. These networks usually have millions of vertices and billions of edges, and each of which has different structures. Many algorithms are designed to work on graph applications which play a vital role in several domains. For instance, Pagerank (PR) is a popular benchmark for graph analytics applications which is used for ordering hyperlinks as well as evaluating sentence similarity [2]. Furthermore, algorithms like Breadth First Search (BFS) and Single-Source Shortest Path (SSSP) are chiefly exercised in cognitive systems embodied various elements from different domains such as machine learning, artificial intelligence, and data analytics. Moreover, Stochastic Gradient Descent (SGD) and Alternating Least Squares (ALS) are two well-known examples of matrix factorization in collaborative filtering primarily utilized for personalized recommendation [3]. Additionally, applications of belief propagation are widely adopted in several domains such as bioinformatics, natural language processing, and pattern recognition in order to solve inference problem by passing local messages between nodes.

Over the past decade, researchers have discussed various types of graph applications for finding shortest paths, optimizing routes, discovering cliques among communities, making targeted product recommendations, clustering, and so forth. For this purpose, they have designed many graph processing frameworks [4–8]. However, each framework optimizes the algorithms by adopting different design decisions. Typically, these decisions determine (1) how to order active nodes for computation, (2) how data moves throughout execution, and (3) how to activate nodes. For instance, Pregel [4] adopts bulk synchronous parallel execution, whereas Graphlab favors asynchronous parallel execution. In terms of information flow, Galois [8] prefers moving data in the push direction, that is to say, an active node performs update operations on its outgoing neighbors. On the other hand, in [9, 10], an active node collects data from its incoming neighbors and updates itself. Moreover, several frameworks utilize a worklist structure for storing nodes activated according to a predefined threshold which need to be processed [8,11,12]. In contrast, other frameworks process each node according to the graph topology without checking its active status [7,10]. As a consequence, due to the aforementioned design choices for a graph algorithm, there is not a complete and yet effective way to implement large-scale parallel graph applications.

Although the computational capabilities of the computers are increased from teraflops to petaflops and beyond, finding an effective implementation of a graph algorithm executed on different types of graphs has many challenges due to the complex patterns of computation in graph analytics applications, large graph sizes, and diversity in graph structures [13]. Not only the properties of the graphs but also the different memory settings have a huge impact on the performance of graph applications [14]. According to recent studies, shared memory implementations show better performance than distributed implementations of graph analytics when graphs fit in the main memory of the system [7, 15, 16]. Communication costs caused by distributed memory settings could not be amortized effectively with an increase in memory bandwidth. Moreover, input graphs with billions of edges could easily fit in the main memory of todays large shared memory machines. Therefore, it is important to understand how parallel graph applications can be implemented effectively on shared-memory systems.

1.1 Objective of the Thesis

This thesis analyzes various combinations of different in-memory implementation styles of graph applications and explores the tradeoffs between performance, scalability, work efficiency, and computation cost of shared-memory systems. We believe that our observations can guide researchers in making effective choices while implementing future graph analytics applications.

Our main contributions can be summarized as follows:

- We implement 8 variants of the PR algorithm, and 4 variants of both SSSP and BFS algorithms with regards to various design decisions such as the order of computations, data access patterns, and work activation.
- We provide performance and scalability analysis for all implementations using both synthetic graphs as well as real social networks.
- We analyze work efficiency of different design choices by taking the total number of edges processed into consideration.
- Finally, a micro-architectural analysis of the algorithms is conducted so as to reveal the bottlenecks of each algorithm that constraint performance of the shared-memory systems.

1.2 Organization of the Thesis

Chapter 2 covers previous work and summarizes existing graph frameworks by considering their characteristics. Chapter 3 reveals main properties of graph applications and examines various design choices for implementing large-scale graph algorithms. Chapter 4 presents different implementations of PR, SSSP, and BFS.

Chapter 5 is divided into 3 sections. Firstly, experimental setup is presented. In this section, we give the summary of datasets used for evaluation, the details of application design, and the system which experiments are executed on. Secondly, performance and scalability analysis for all applications are given. Their speedups with respect to the running time of a baseline sequential algorithm are also reported. As work efficiency is an important metric, we give the work efficiency of each application. In the third section, the performance of the system is evaluated by using hardware performance counters.

Chapter 6 discusses our observations and conclusions from the experimental results. Finally, we conclude the thesis with a brief overview of our work, accomplishments, and future remarks.

Chapter 2

Related Work

Recently, graph analytics applications have received considerable attention from both industry and academic environments. Many frameworks eliminate major difficulties of graph processing through various abstractions. In this chapter, a short review of those frameworks will be given to illustrate their main differences according to various design choices such as the order of computations, data access patterns, and work activation.

An important design consideration is the target platform while choosing an appropriate framework. While some frameworks target distributed architectures, others focus on shared memory systems. Furthermore, some frameworks allow users to make decisions about data flow (pull or push) and execution order (synchronous or asynchronous). However, other systems might impose a limit to utilize only a single model. For instance, Pregel [4] uses push-based implementation, while GraphLab [5] uses pull-based. Additionally, some frameworks such as Galois [8] and GraphChi [17] are restricted to asynchronous execution, while GraphLab [5] lets the user decide during implementation. Apart from the design selections regarding information flow and execution order, most frameworks offer a vertex-centric model for programming, although some others such as Ligra [7] uses graph-centric model. Table 2.1 lists different frameworks with their properties including execution model, flow model, and programming model.

There is no clear consensus among the graph processing frameworks on optimum design selections, thus benchmarks used for these systems depend on the input data and design decisions that developer enforces. Proper benchmark selection is critical in comparing frameworks based on performance criteria such as runtime or scalability. There are graph processing benchmark suites such as GAP [10] and CRONO [18] to encourage the standardization of graph processing evaluations. Table 2.1 summarizes aforementioned graph frameworks with their characteristics.

Table 2.1: Summary of frameworks used for graph processing.

Framework	***Year***	***System***	***Execution Model***	***Flow Model***	***Programming Model***
Pregel [4]	2010	Distributed	Synchronous	Push	Vertex-Centric
GraphLab [5]	2010-12	Both	Both	Both	Vertex-Centric
GraphChi [17]	2012	Shared	Asynchronous	Pull	Vertex-Centric
Giraph [19]	2012	Distributed	Asynchronous	Push	Vertex-Centric
Galois [8]	2013	Shared	Synchronous	Both	Vertex-Centric
Ligra [7]	2013	Shared	Asynchronous	Push	Graph-Centric
GPS [20]	2013	Distributed	Synchronous	Push	Vertex-Centric

We focus on in-memory implementations of graph algorithms. However, there exist many graph frameworks executed on Flash/SSD platforms such as GraphCHi [17], TurboGraph [21], X-Stream [22], FlashGraph [23], and GridGraph [24]. Moreover, some graph frameworks like Medusa [25], Gunrock [26], and CuSha [27] are designed to work on GPU systems. There are also some attempts to design specific accelerators for graph processing such as Ozdal et al. [28], GraphOps [29], and Tesseract [30].

Chapter 3

Principal Design Decisions for Graph Applications

In this chapter, we discuss main characteristics of graph analytics applications and examine several alternatives of design choices for implementing graph algorithms.

3.1 Properties of Graph Applications

Graph analytics applications have received considerable attention recently due to its applicability to different domains such as web graphs, social networks, and protein networks. Designing a large scale efficient graph processing system is a challenging task due to the large volume and the irregularity of communication between computations at each vertex or edge. As a result, several graph analytics frameworks [4, 5, 8, 19, 31, 32] adopting different programming models have been developed.

Graph frameworks process graphs either synchronously or asynchronously. The former is easy to program since it processes data simultaneously and iteratively; whereas, the latter needs to order data updates carefully using latest available

dependent state. Moreover, many graph applications are executed until a convergence criterion is satisfied. This behavior is implemented simply by executing all vertices iteratively until the convergence is reached. However, there is no need to execute all vertices in every iteration because some vertices may converge faster than others, causing asymmetric convergence.

We focus on graph algorithms that display three main characteristics:

- We consider vertex-centric algorithms rather than edge-centric or graph-centric. In such applications, a vertex performs local computation on its neighbor vertices (e.g., incoming vertices or outgoing vertices). However, during computation, overlapping neighborhoods might become a bottleneck since they typically need synchronization between threads.
- For each vertex an iterator iterates through its incoming and/or outgoing edges.
- They are iterative and perform a local computation at a set of vertices repeatedly until a convergence criterion is met.

Furthermore, vertex-centric executions can be specialized for different applications. For this purpose, we classify design choices in three orthogonal dimensions as shown in Figure 3.1.

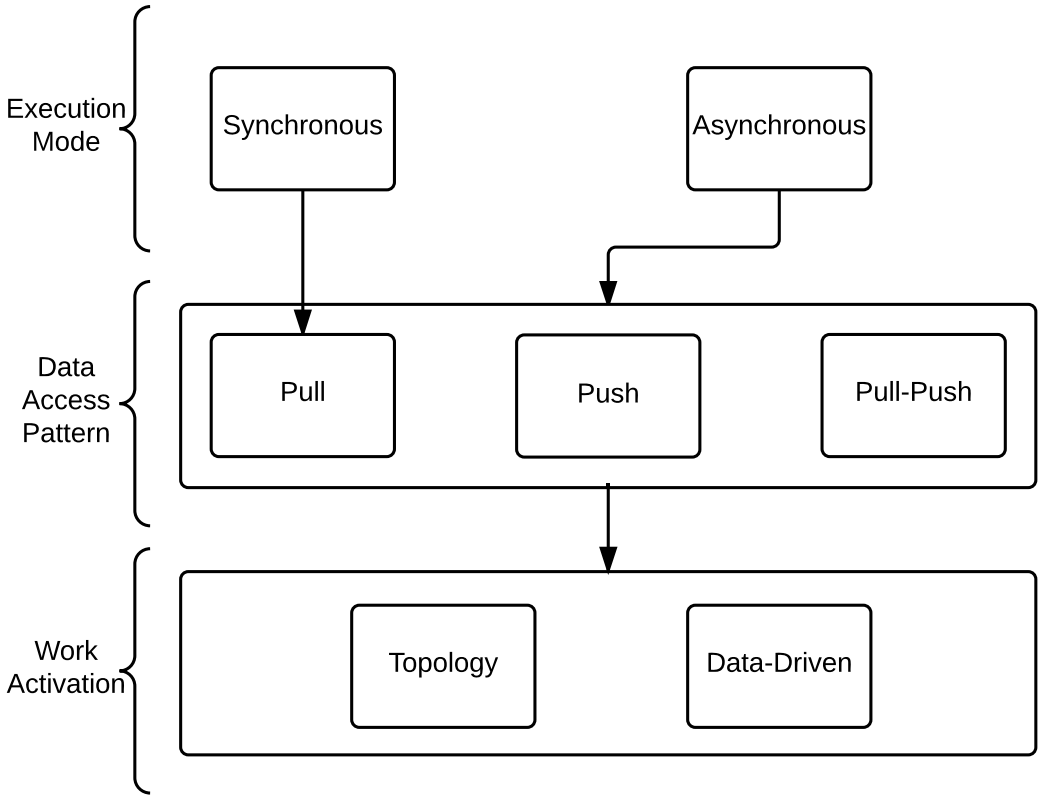

Figure 3.1: Algorithm design choices in graph applications.

When applications are designed in a data-driven manner, two additional design choices need to be considered, namely, the structure of a frontier which tracks active nodes and scheduling of frontiers. However, we do not consider these data-driven choices, rather we focus on the first three dimensions. In the following three sections, we describe the aforementioned dimensions in detail.

3.2 Order of Computations

The first dimension is to decide whether to use fine-grain or coarse-grain synchronization. One option is synchronous execution that provides synchronization between its iterations by placing barriers. This method eliminates fine grain synchronization such as using locks for every vertex or applying atomic operations. The second method employs fine grain synchronization in order to implement applications asynchronously. An algorithm executed asynchronously needs to perform update operations on the vertices and edges atomically so that sequential consistency will be protected.

In order to avoid any conflict while updating the value of each vertex in parallel, we can use an atomic exchange function as shown in Algorithm 3.1. The function takes three parameters such as an array pointer, an index value, and the desired value. The array pointer and the index value are used for pointing the atomic object which is expected to be updated with the desired value. If the value pointed by the atomic object (e.g., expected[index]) and the non-atomic expected value (e.g., oldVal) are equal, the function atomically exchanges the value of the atomic object with the desired value. Otherwise, the function atomically loads the old value of the atomic object. Therefore, we need an additional statement for ensuring correct exchange operation.

Algorithm 3.1 Atomic Exchange

```
1: function ATOMICEXCHANGE(*expected, index, desired)
2:     oldVal = expected[index]
3:     while !expected[index].compare_exchange_weak(oldVal,newVal) do
4:         oldVal = expected[index]
5:     end while
6: end function
```

Moreover, built-in functions like compare-and-swap (CAS), atomic-min, fetch-and-add can be used for accessing memory atomically. We used atomic exchange and CAS functions in our implementations.

3.3 Data Access Patterns

The second dimension is to select an appropriate data access pattern in which we have three different choices. The first one is a pull-based implementation which indicates the direction of the data movement. An algorithm implemented in the pull direction iterates over incoming (or outgoing) edges to gather their data and executes a reduction with neighbors' data. Note that, this is only a read operation. On the contrary, in push-based implementations, neighbors are updated by the vertex being processed. These write operations are implemented with atomic operations such as compare-and-swap (CAS) primitive. And finally, the pull-push operation has both of the characteristics of pull and push. This implementation iterates over neighbors and reduces their data with atomic operations.

3.4 Work Activation

The third dimension is to select whether to process all vertices at every iteration or only process those vertices with updates. In terms of work activation, implementations are classified into two main types: topology-driven or data-driven.

Topology driven implementation pretends that all graph vertices are active, thus processing each node in every iteration without considering whether some nodes have updates or not. As expected, this causes more computation and increased irregular memory accesses, thereby causing inefficiencies. This is more critical especially for graphs which embodies large sparsity. Despite this drawback, topology driven implementations eradicate the worklist usage for activation.

On the other hand, the data-driven model keeps a list of active vertices which are recently updated. Therefore, the algorithm visits the nodes and does the computation according to whether they are on the list or not. This optimization typically prevents unnecessary computations and memory accesses. Data-driven approach is preferable for multicore programming in order to ameliorate the work efficiency. However, the management of work list structure is challenging.

Chapter 4

Implementations of Selected Graph Algorithms

We implemented 8 different versions of PR algorithm by considering 3 design choices as explained: the order of computations, data access pattern, and work activation. Synchronous techniques used in graph traversal are generally expected to cause performance loss due to load imbalance between bariers [33] when compared to their asychronous counterparts. Also, pull-push based methods are not suitable for SSSP and BFS algorithms. For these reasons, we implement 4 different versions of SSSP and BFS by considering different combinations of data access patterns and work activations. These applications are designed to be executed asynchronously due to performance bahaviors. In this chapter, we describe the details of these applications with different parameters.

4.1 Pagerank (*PR*)

PR is a widely adopted benchmark in many frameworks [7,8,10,15,16,34] since it displays the properties of graph applications as mentioned in the previous chapter. In addition, it captures the irregular memory access, work scheduling, and load

imbalance characteristics of many graph algorithms. Algorithm 4.1 gives the details of base PR algorithm.

Algorithm 4.1 Topology-driven Pull-based Synchronous PR

Input: G = (V,E)
Output: scores

```
1:  scores[:] = 1.0 − α
2:  for i until maxIter do
3:      activeNum = 0
4:      parfor v ∈ V do
5:          sum = 0
6:          for w ∈ inNeighbor(v) do
7:              sum += scores[w]/outDegree(w)
8:          end for
9:          nextScores[v]=(1.0 − α) + α*sum
10:         if fabs(nextScores[v] - scores[v]) ≥ ε) then
11:             activeNum++
12:         end if
13:     end parfor
14:     swap(scores, nextScores)
15:     if activeNum ≤ 0 then
16:         break
17:     end if
18: end for
19: scores = scores / | V |
```

In terms of execution, PR can be performed both synchronously and asynchronously. As shown in Equation 4.1, power method [2] can be employed in order to calculate rank values in a synchronous manner. This method holds both current and previous ranks of each vertex. In each iteration, a vertex calculates its new rank by using ranks which are calculated in the previous iteration.

$$Pr^{t+1}[u] = \alpha \times \sum_{w \in w \to u} \frac{Pr^{t}[w]}{T_w} + (1 - \alpha) \tag{4.1}$$

On the other hand, PR algorithm can be performed asynchronously by employing Gauss-Seidel method [35] as shown in Equation 4.2 In this case, each vertex updates its rank by utilizing the most recent ranks calculated. For each vertex,

only the most up-to-date rank is stored and iterations are not separated by barriers; therefore, iterations are not clearly defined in asynchronous applications. In parallel implementations, PR algorithm implemented by using Gauss-Seidel formulation should synchronize threads because one thread might try to write a recently calculated rank for a vertex while a different thread might try to read the rank of the same vertex.

$$Pr^{t+1}[u] = \alpha \times \left(\sum_{w \in w \to u} \frac{Pr^{t}[w]}{T_w} + \sum_{v \in v \to u} \frac{Pr^{t+1}[v]}{T_v} \right) + (1 - \alpha) \tag{4.2}$$

In terms of data access patterns, PR applications can be implemented in three different ways, namely pull, push, and pull-push. Pull-based PR can be implemented both synchronously and asynchronously; however, push-based and pull-push based versions should be implemented by Gauss-Seidel [35]. Synchronous pull-based applications utilize two lists for keeping current and next ranks of each vertex, and in each iteration, a vertex can only see the ranks of its in-neighbors from previous iterations. Moreover, each vertex pulls the ranks of its incoming neighbors and updates itself only once per step. Therefore, pull-based applications do not require synchronization since the elements of the list are updated only once in an iteration. In contrast, asynchronous pull-based applications use only one list in order to keep ranks of each vertex. For this reason, they need an exclusive lock for writing and updating data since more than one write is issued for some vertices at the same time.

From work activation perspective, PR applications can be implemented in both styles such as topology-driven and data-driven. We combined these two work activation schemes with all data-access patterns as mentioned before. Algorithms 4.3 and 4.4 illustrate how an active node accesses its neighbours in the push direction and pull-push direction, respectively. As can be seen in Algorithm 4.4, after an active node accesses its in-neighbors to gather their contributions, the active node updates its out-neighbors' rank immediately. Therefore, total work done in pull-push based method, is augmented. Furthermore, using a worklist, we

track each active vertex by utilizing bit vector structures in data-driven applications. As an alternative, we also implemented a worklist based on a central queue structure by using lock primitives. However, in our experiments, both worklist structures exhibit similar performance. For this reason, we only report the results for bit vector implementation.

Algorithm 4.2 Data-Driven Pull-based Asynchronous PR

```
Input: G = (V,E)
Output: ranks
aRanks[:] = 1.0 − α
frontier[:] = true
next[:] = false
for i until maxIter do
    activeNum = 0
    parfor v ∈ V do
        if frontier[v] then
            sum = 0
            for w ∈ inNeighbor(v) do
                sum += aRanks[w]/outDegree(w)
            end for
            oldRank = aRanks[v]
            newRank =(1.0 − α) + α*sum
            atomicExchange(aRanks, v, newRank)
            if fabs(newRank - oldRank) ≥ ε) then
                for w ∈ outNeighbor(v) do
                    if !next[w] then
                        next[w] = true
                    end if
                end for
                activeNum++
            end if
        end if
    end parfor
    swap(frontier, next)
    next[:] = false
    if activeNum ≤ 0 then
        break
    end if
end for
ranks = aRanks / | V |
```

Algorithm 4.3 Data-Driven Push-based Asynchronous PR

```
Input: G = (V,E)
Output: ranks
parfor v ∈ V do
    ranks[v] = 1.0 − α
    aResiduals[v] = 0.0
    frontier[v] = true
    next[v] = false
    for w ∈ inNeighbor(v) do
        aResiduals[v] + = 1.0/ outDegree(w)
    end for
    aResiduals[v] = (1.0 − α) ∗ α∗aResiduals[v]
end parfor
for i until maxIter do
    activeNum = 0
    parfor v ∈ V do
        if frontier[v] then
            ranks[v] += aResiduals[v]
            delta = α∗(aResiduals[v]/outDegree(v))
            for w ∈ outNeighbor(v) do
                oldRes = aResiduals[w]
                atomicExchange(aResiduals, w, oldRes + delta)
                if fabs(oldRes + delta ≥ ε&& oldRes ≤ ε) then
                    activeNum++
                    if !next[w] then
                        next[w] = true
                    end if
                end if
            end for
            aResiduals[v] = 0.0
        end if
    end parfor
    swap(frontier, next)
    next[:] = false
    if activeNum ≤ 0 then
        break
    end if
end for
ranks = ranks / | V |
```

Algorithm 4.4 Data-Driven Pull-Push-based Asynchronous PR

Input: G = (V,E)
Output: ranks

```
parfor v ∈ V  do
    aRanks[v] = 1.0 − α
    aResiduals[v] = 0.0
    frontier[v] = true, next[v] = false
    for w ∈ inNeighbor(v) do
        aResiduals[v] + = 1.0/ outDegree(w)
    end for
    aResiduals[v] = (1.0 − α) ∗ α∗aResiduals[v]
end parfor
for i until maxIter do
    activeNum = 0
    parfor v ∈ V  do
        if frontier[v] then
            sum = 0
            for  w ∈ inNeighbor(v) do
                sum += aRanks[w]/outDegree(w)
            end for
            newRank = (1.0 − α + α∗)sum
            atomicExchange(aRanks, v, newRank)
            delta = α∗(aResiduals[v]/outDegree(v))
            for  w ∈ outNeighbor(v) do
                oldRes = aResiduals[w]
                atomicExchange(aResiduals, w, oldRes + delta)
                if  fabs(oldRes + delta ≥ ε&& oldRes ≤ ε) then
                    activeNum++
                    if !next[w] then
                        next[w] = true
                    end if
                end if
            end for
            aResiduals[v] = 0.0
        end if
    end parfor
    swap(frontier, next)
    next[:] = false
    if activeNum ≤ 0 then
        break
    end if
end for
ranks = aRanks / | V |
```

4.2 Single-Source Shortest Path (*SSSP*)

In the single source shortest path (SSSP) problem, we try to find minimum cost path from a single source node to all other nodes in a weighted directed graph. A well-known sequential algorithm to solve the SSSP problem is Dijkstra's algorithm [36]. A parallel algorithm, called delta-stepping [37, 38], divides Dijkstra's algorithm into buckets which can be executed in parallel. The alternative parallel algorithm for solving the SSSP problem is Bellman-Ford [39] which allows negative edge weights. Our SSSP implementations are adopted from this algorithm.

We implemented four different versions of SSSP algorithm by considering different data access patterns and work activation choices. Note that, SSSP is implemented asychronously since their synchronous implementations are expected to show poor perforamance as explained in literature [33, 38]. As shown in Algorithm 4.5, in pull-based applications, an active node updates its distance by pulling (reading) data from its incoming neighbors. The write operation is only performed on the active node. However, in push-based applications, data flows from the active node to its outgoing neighbors. Since the active node updates its outgoing neighbors distances by pushing its distance value to its outgoing neighbors, push-based applications generate more frequent updates.

We provide the pseudo code for both pull-based and push-based versions as shown in Algorihtms 4.5 and 4.6, respectively. Both of them employ bit vectors in order to keep track of active vertices for each iteration. Each algorithm iterates until there is no change in the distance value of any node. As explained before, vertex updates are performed asynchronously.

Algorithm 4.5 Data-Driven Pull-based SSSP

Input: G = (V,E), source, w
Output: dists

```
aDists[:] = +∞
frontier[:], next[:] = false
dists[source] = 0
frontier[source] = true
for v ∈ outNeighbor(source) do
    frontier[v] = true
end for
for i until maxIter do
    activeNum = 0
    parfor u ∈ V do
        minDist = +∞
        for v ∈ inNeighbor(u) do
            if minDist > aDists[v] + w(v,u) then
                minDist = aDists[v] + w(v,u)
            end if
        end for
        if aDist[u] > minDist then
            atomicExchange(aDist, u, minDist)
            for v ∈ outNeighbor(u) do
                if !next[v] then
                    next[v] = true
                end if
            end for
            activeNum++
        end if
    end parfor
    swap(frontier, next)
    next[:] = false
    if activeNum ≤ 0 then
        break
    end if
    if activeNum ≤ 0 then
        break
    end if
end for
dist = aDists
```

Algorithm 4.6 Data-Driven Push-based SSSP

Input: G = (V,E), source, w
Output: dists

```
aDists[:] = +∞
frontier[:], next[:] = false
aDists[source] = 0
frontier[source] = true
for i until maxIter do
    activeNum = 0
    parfor u ∈ V do
        if frontier[u] then
            for v ∈ outNeighbor(u) do
                if aDists[v] > aDists[u] + w(u,v) then
                    newDist = aDists[u] + w(u,v)
                    atomicExchange(aDists, v, newDist )
                    activeNum++
                    if !next[v] then
                        next[v] = true
                    end if
                end if
            end for
        end if
    end parfor
    swap(frontier, next)
    next[:] = false
    if activeNum ≤ 0 then
        break
    end if
end for
dist = aDists
```

4.3 Breadth-First Search(*BFS*)

As a well-known algorithm, in breadth-first search (BFS), the goal is to find the breadth-first order traversal of the graph vertices. BFS is similar to SSSP in which edge weights are set to be 1 instead of using weights from the input. Similar to SSSP, our BFS implementations also follow the logic in Bellman-Ford [39].

We have implemented four different versions of BFS by considering two different design choices, namely, data access pattern and work activation. As shown in Algorithm 4.7, pull-based version executes a reduction over incoming edges and finds the minimum level of neighbors, then corresponding vertex level is updated accordingly. On the other hand, Algorithm 4.8 illustrates a push-based implementation where outgoing neighbors update their level by using atomic operations. Different BFS implementations follow the same pattern as in SSSP, except the fact that edge weight is always 1. Moreover, BFS applications are designed to be executed asynchronously since recent studies show that asynchronous implementations show better performance than their synchronous counterparts [40].

Algorithm 4.7 Data-Driven Pull-based BFS

Input: G = (V,E), source
Output: levels

```
aLevels[:] = +∞
frontier[:], next[:] = false
dists[source] = 0
frontier[source] = true
for v ∈ outNeighbor(source) do
    frontier[v] = true
end for
for i until maxIter do
    activeNum = 0
    parfor u ∈ V do
        minLevel = +∞
        for v ∈ inNeighbor(u) do
            if minLevel > aLevels[v] + 1 then
                minLevel = aLevels[v] + 1
            end if
        end for
        if aLevels[u] > minLevel then
            atomicExchange(aLevels, u, minLevel)
            for v ∈ outNeighbor(u) do
                if !next[v] then
                    next[v] = true
                end if
            end for
            activeNum++
        end if
    end parfor
    swap(frontier, next)
    next[:] = false
    if activeNum ≤ 0 then
        break
    end if
    if activeNum ≤ 0 then
        break
    end if
end for
levels = aLevels
```

Algorithm 4.8 Data-Driven Push-based BFS

Input: G = (V,E), source
Output: aLevels

```
aLevels[:] = +∞
frontier[:], next[:] = false
aLevels[source] = 0
frontier[source] = true
for i until maxIter do
    activeNum = 0
    parfor u ∈ V do
        if frontier[u] then
            for v ∈ outNeighbor(u) do
                if aLevels[v] > aLevels[u] + 1 then
                    newLevel = aLevels[u] + 1
                    atomicExchange(aLevels, v, newLevel)
                    activeNum++
                    if !next[v] then
                        next[v] = true
                    end if
                end if
            end for
        end if
    end parfor
    swap(frontier, next)
    next[:] = false
    if activeNum ≤ 0 then
        break
    end if
end for
levels = aLevels
```

Chapter 5

Experiments

5.1 Experimental Setup

We conducted our experiments on a multi-socket server system with specifications given in Table 5.1. All algorithms are implemented using C++ and OpenMP, and the datasets are stored in Compressed Sparse Row (CSR) format [41]. For each vertex, we store their in-neighbors and out-neighbors in separate sets in order to improve locality. The runtime is the average results of 10 runs, and it includes the time for allocation and initialization of all data structures for each algorithm (except initial graph itself). We use Performance API (PAPI) [42] in order to access hardware counters in the system, thereby observing the characteristics of underlying architecture.

Table 5.1: System details for our experiments.

Component	**Specification**
CPU	Intel Xeon E5-2643 @3.30GHz, 8 cores, 2 sockets, 4 cores/socket, 2 threads/core
Cache	Private 32 KB L1 cache, Private 256KB L2 caches Shared 10MB L3 cache
Main Memory	264GB

We evaluate the implementations of all design alternatives using the datasets in Table 5.2. Our testbed involves both small graphs as well as large graphs. For our synthetic data sets, RMAT graph is generated with parameters (A,B,C) = (0.45, 0.25, 0.15) by using Graph500 benchmarks [43]. Pay Level Domain is a hyperlink graph obtained from the Common Crawl web corpora [44]. We select Google Web Graph (wg), and soc-LiveJournal (lj) from the SNAP datasets [45]. All graphs are directed and duplicate edges are removed.

Table 5.2: Datasets used for evaluation.

Dataset	*Abv.*	*#Vertices*	*# Edges*	*Degree*	*Directed*
Google Web Graph	wg	916K	5.1M	8.1	Y
LiveJournal	lj	4.8M	68.9M	6.5	Y
Pay Level Domain	pld	33.6M	623.06M	14	Y
RMAT	rmat25	42.9M	536.84M	15	Y

In this work, we implement 16 different versions of 3 algorithms by considering different design decisions. The detailed summary of design choices for each application is found in Table 5.3.

Table 5.3: Summary of application test cases.

Algorithm	*Application*	*# Activation*	*# Access*	*Execution*
pr	tp_pull_syn	topology	pull	synchronous
pr, sssp, bfs	tp_pull_asyn	topology	pull	asynchronous
pr, sssp, bfs	tp_push	topology	push	asynchronous
pr	td_pull_push	topology	pull-push	asynchronous
pr	dd_pull_syn	data-driven	pull	synchronous
pr, sssp, bfs	dd_pull_asyn	data-driven	pull	asynchronous
pr, sssp, bfs	dd_push	data-driven	push	asynchronous
pr	dd_pull_push	data-driven	pull-push	asynchronous

5.2 Performance Results

This section presents performance analysis of the implementations in terms of execution time and scalability. Moreover, we report speedups of the algorithms relative to the best serial execution in each dataset.

5.2.1 Runtime and Scalability

5.2.1.1 PR

Figures 5.1-4 show the performance of different implementations of PR in terms of execution time with respect to the number of threads. These figures illustrate that among all implementations, topology driven pull-push-based methods have poor performance when compared to both pull-based and push-based methods. This is expected as pull-push-based methods require reading operations on in-neighbors as well as write operations on out-neighbors. However, in pull-based methods, a vertex performs only one update operation on itself and performs many read operations on its incoming edges. Likewise, push-based methods perform only write operations on itself and its out-neighbors instead of requiring both read and write operations on its neighbors as in pull-push-based methods.

Since pull based methods require only one write and many read operations, they are expected to be more cache friendly. Nevertheless, as shown in Figures 5.1-4, topology driven pull-based methods give the second-worst performance among all implementation styles. Although push-based methods require more frequent write operations, data-driven push-based methods are the fastest among all implementation variants for all datasets. One reason for such behavior is that push-based methods accelerate the rate of the dissemination of information, hence they can amortize the overhead required for frequent writes. However, this is not the case for topology driven push-based methods. Figures 5.1 and 5.2 show that quick frequent updates on ranks do not improve the performance of topology driven push-based methods sufficiently because they need to iterate

over all the nodes without checking their activation status. As a matter of fact, these frequent updates increase the number of nodes activated in topology based implementations of other algorithms such as SSSP and BFS.

Our observations also show that the performance of data-driven methods is better than the performance of topology driven ones. This result is also consistent with the observations in [46]. Data-driven methods employ a worklist in order to keep active nodes whose ranks change more from a defined threshold, hence they can filter out many edges instead of processing them unnecessarily. As can be seen in Figures 5.1-4, the effectiveness of employing a frontier for tracking active nodes can be clearly seen by looking at the performance improvement of the pull-push based method. When utilizing an active list, the run time of the algorithms decreases drastically compared to topology driven ones as illustrated in Figure 5.3.

Next, we consider the choice of executing the applications with asynchronous execution model. The results demonstrate that if we execute a method asynchronously, its performance improves. Although asynchronous executions require atomic operations, they are able to mitigate synchronization overheads by accelerating convergence rate of the algorithm. For instance, pull-based asynchronous applications perform better compared to their synchronous counterparts on large graphs as shown in Figures 5.3 and 5.4. However, on small graphs, Figures 5.1 and 5.2 show that only their topology-driven pull-based versions outperform their synchronous versions. When applications use a worklist for tracking active nodes, synchronous versions deliver higher performance than their asynchronous counterparts on small graphs. One possible drawback of synchronous execution is due to the slow convergence rate. In each iteration, a vertex can only see the update from previous iteration, hence the information is disseminated more slowly when compared to the methods executed in Gauss-Seidel way. In asynchronous models, the order of updates is not separated with barriers and only one global list is used for keeping rank values. As a result, each vertex can detect the most up-to-date version of data which increases the convergence rate of the algorithms.

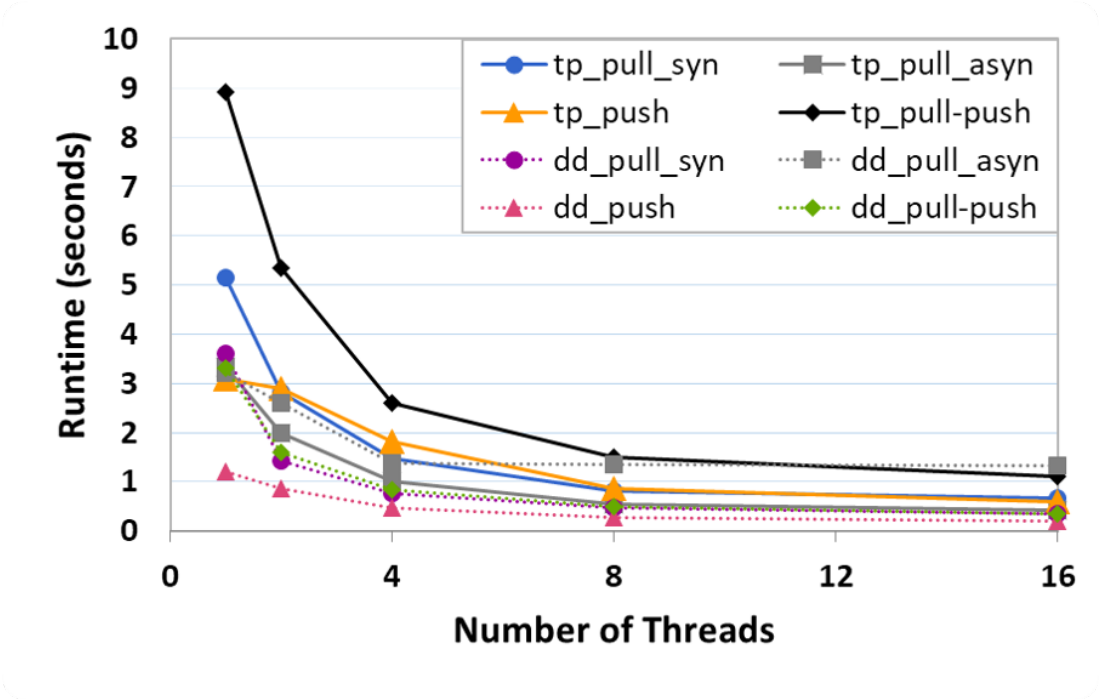

Figure 5.1: Execution time of PR with *wg* dataset.

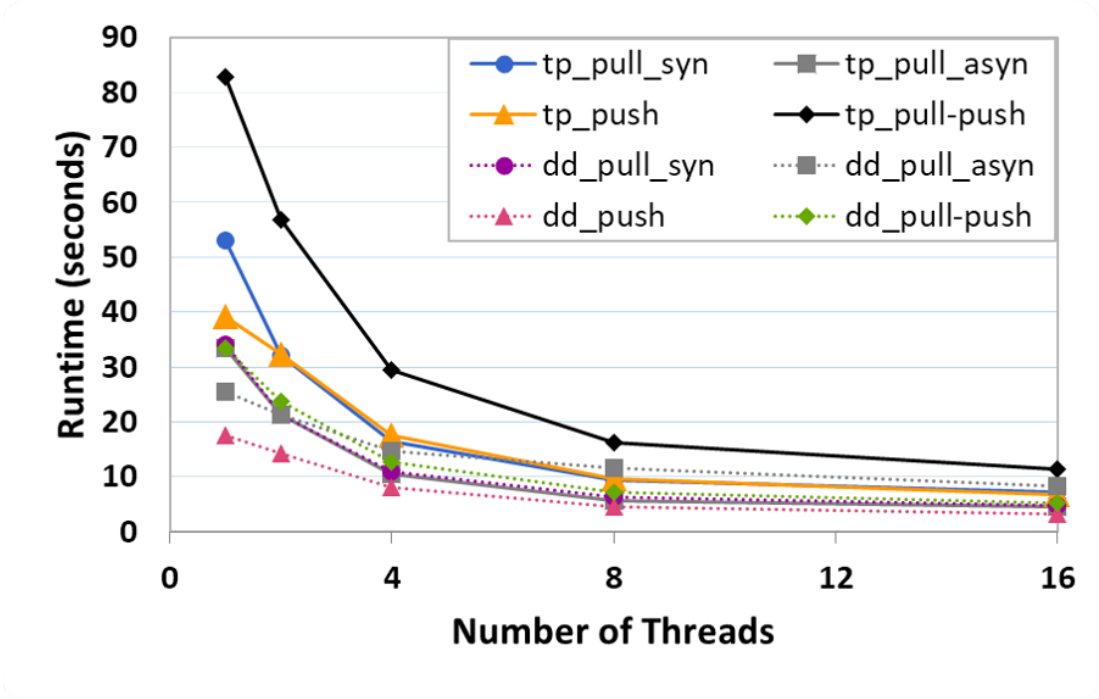

Figure 5.2: Execution time of PR with *lj* dataset.

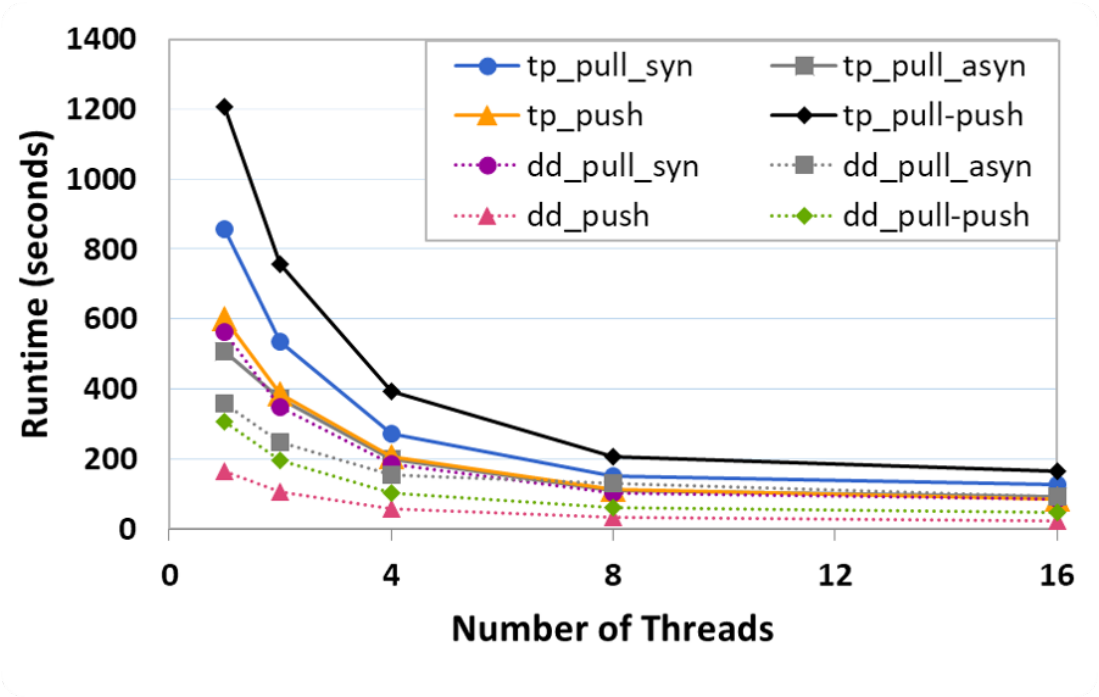

Figure 5.3: Execution time of PR with *pld* dataset.

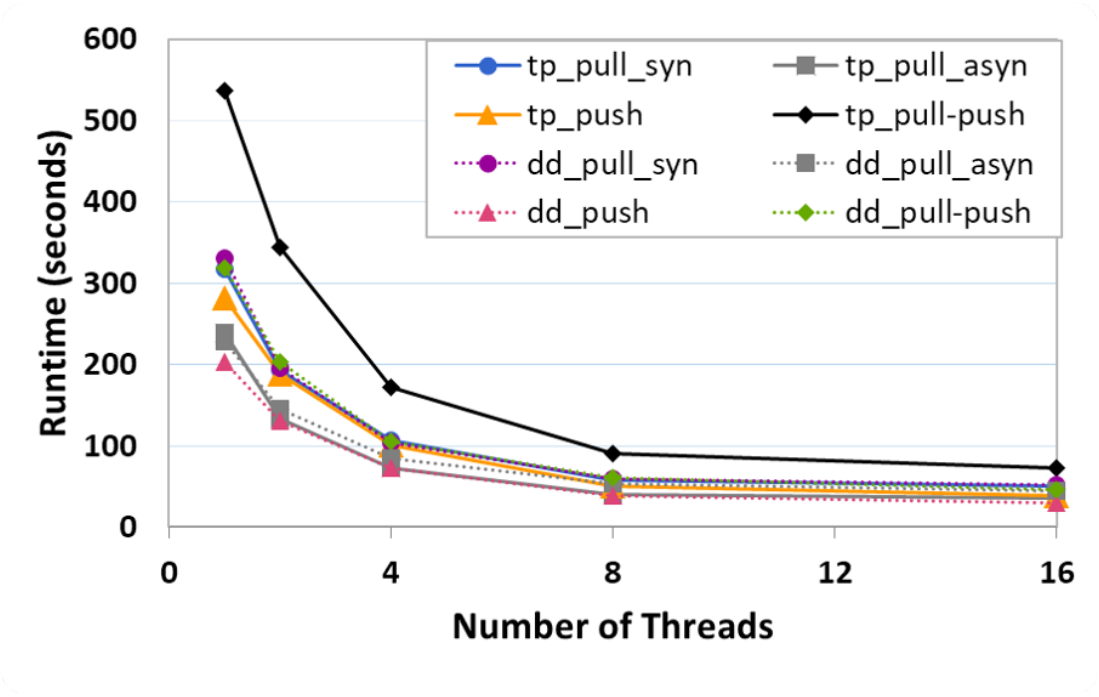

Figure 5.4: Execution time of PR with *rmat* dataset.

Scalability is used as another performance metric which measures a parallel system's capacity to increase speedup with respect to the number of threads. For this purpose, the scalability of different implementations of PR is given in Figures 5.5-8. In these figures, Y-axis indicates self-relative scalability, and X-axis represents the number of threads. For all datasets, asynchronous data-driven pull-based methods perform the poorest scalability although other alternatives of pull-based methods scale very well. Both synchronous and asynchronous variants of topology driven pull-based methods exhibit good scalability. However, for data-driven pull-based methods, asynchronous versions have much less scalability than the synchronous ones as shown in Figures 5.5-8. This indicates that mode of execution becomes significant when the algorithm is implemented in a data-driven way.

As can be seen in Figures 5.7 and 5.8, for large graphs, all classes of algorithms except the asynchronous data-driven pull-based ones are highly scalable. Interestingly, although the topology-driven pull-push-based methods display the poorest performance, they show the best scalability. Moreover, data-driven push-based methods show slightly less scalability for *pld* graph. For small graphs, we observe a different pattern in scalability when compared to the large graphs. Push-based methods show the second-poorest scalability. One possible reason is the fact that push-based methods require frequent write operations on out-neighbors, thereby limiting the scalability. On the other hand, the scalability of pull-push based methods are significantly high, especially for large graphs as shown in Figures 5.7 and 5.8.

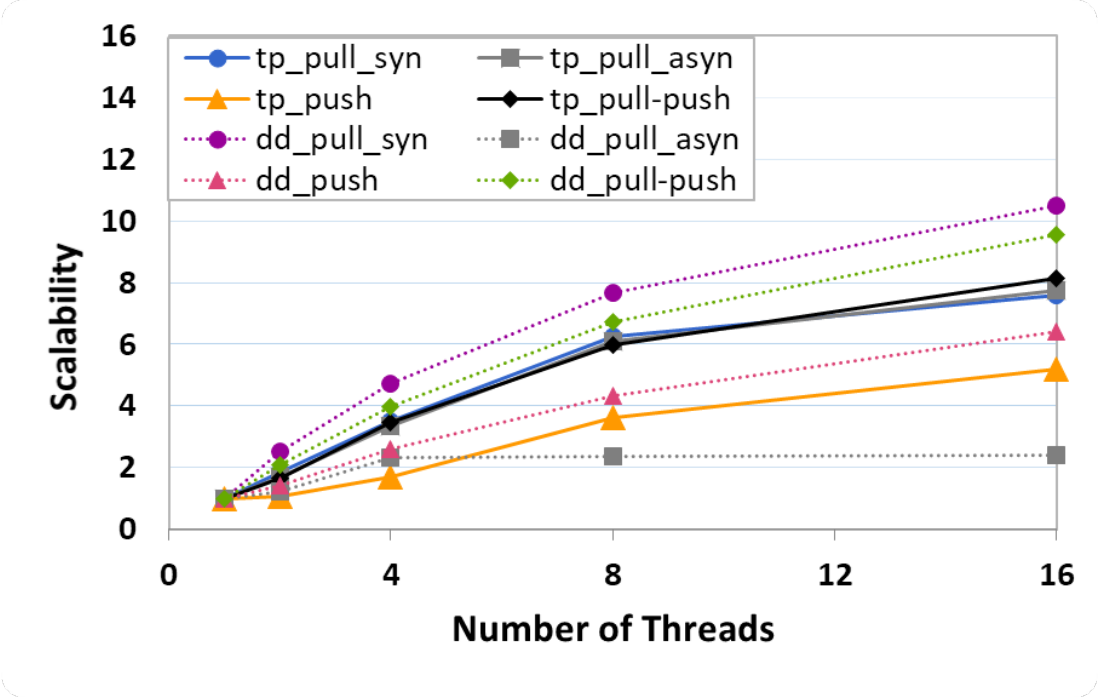

Figure 5.5: Scalability of PR with *wg* dataset.

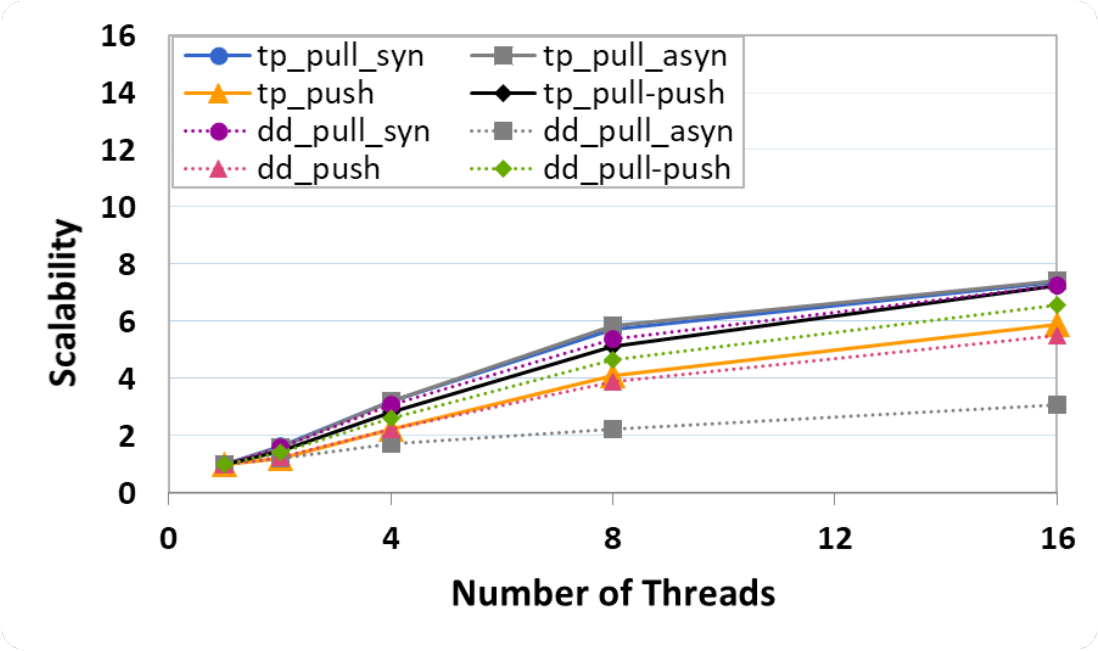

Figure 5.6: Scalability of PR with *lj* dataset.

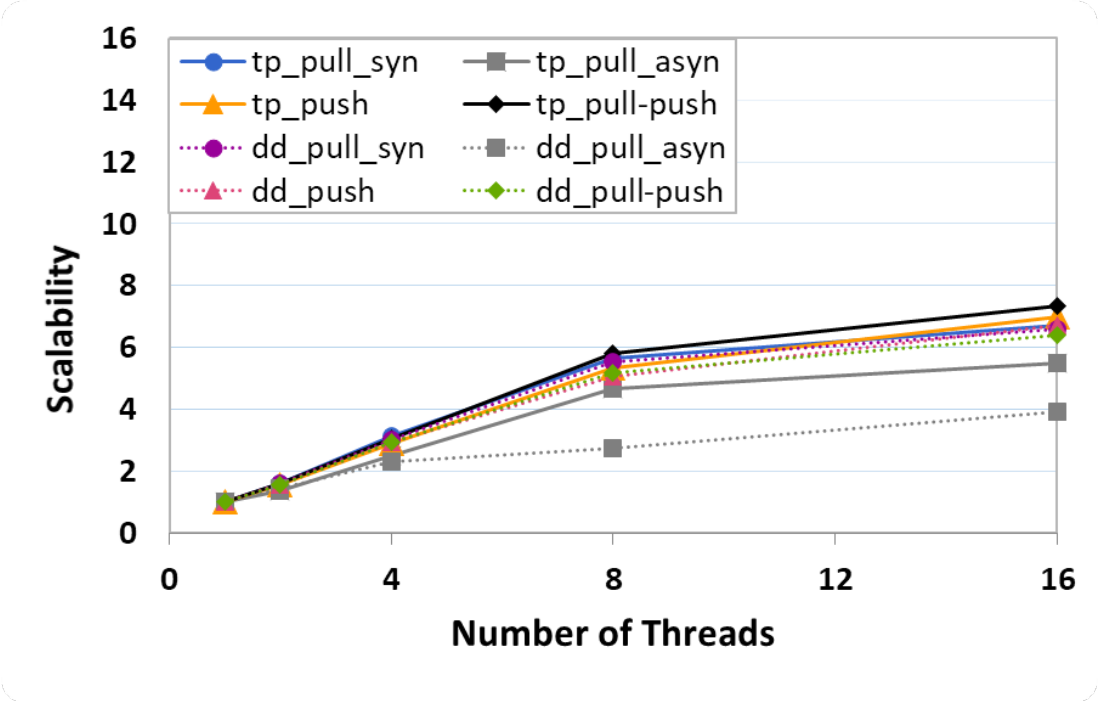

Figure 5.7: Scalability of PR with *pld* dataset.

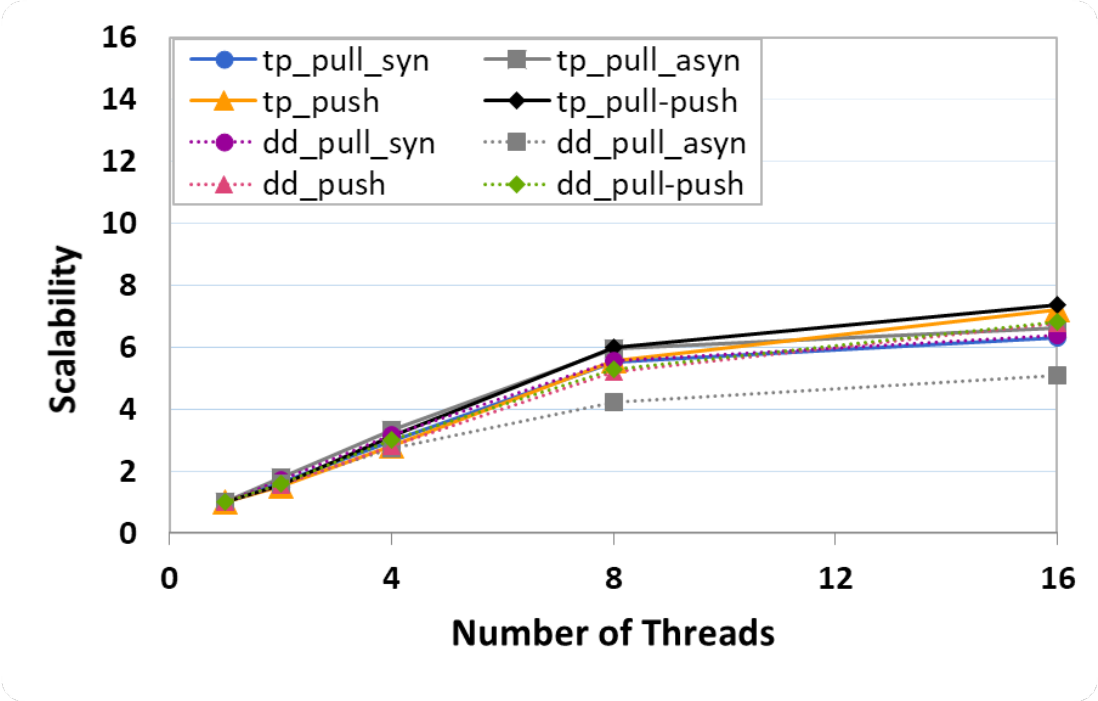

Figure 5.8: Scalability of PR with *rmat* dataset.

5.2.1.2 SSSP

Figures 5.9-10 and 5.11-12 indicate execution time and scalability of different imlementations of SSSP algorithm. Among all variants of SSSP, topology-driven based methods show less performance than data-driven methods as shown in Figures 5.9 and 5.10. By contrast, the implementations which employ topology-based activation scheme are highly scalable. We observe that data-driven implementations are also scalable but their scalability is low compared to the topology-driven alternatives. Although data-driven push-based methods are faster than topology-driven pull based methods, both methods perform similar scalability.

For different data access patterns, work activation plays a significant role in the performance and scalability of the algorithms. For push-methods, activation scheme has a huge impact on both performance and scalability. If nodes are activated according to the structure of a graph, then the algorithm shows less performance but high scalability. In contrast, if nodes are activated by utilizing a frontier, the algorithm has better performance but it shows relatively low scalability. This observation is valid for all datasets except *rmat* graph as shown in Figure 5.12. All variants of SSSP algorithm demonstrates similar scalability with respect to one another on *rmat* graph.

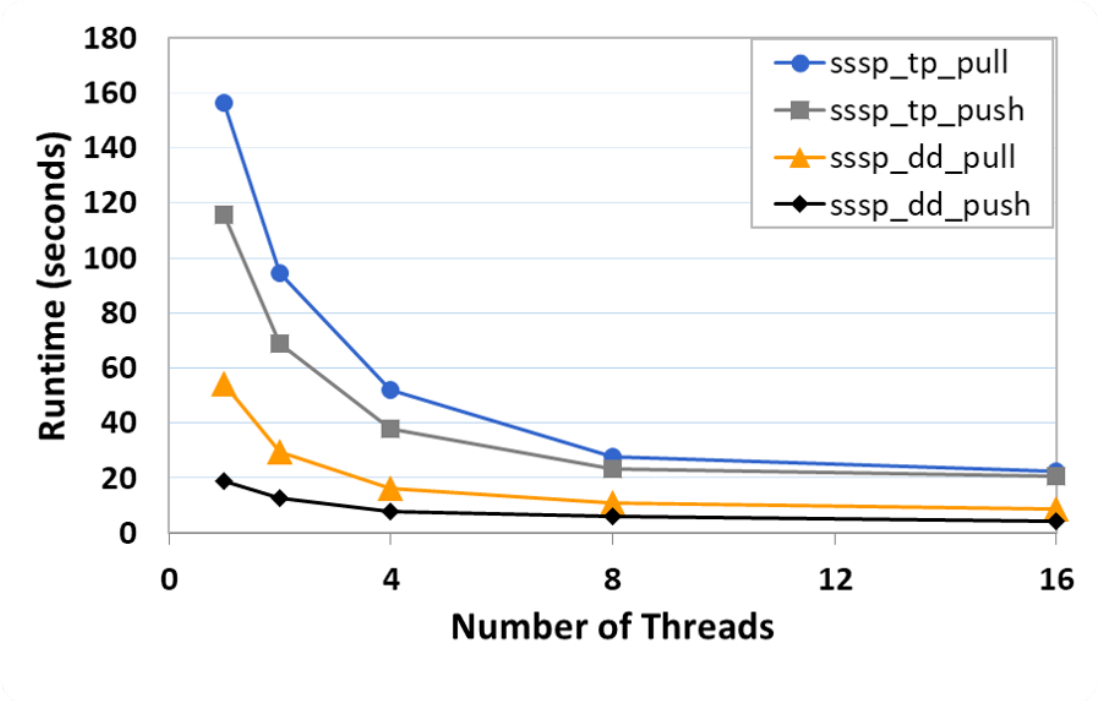

Figure 5.9: Execution time of SSSP with *pld* dataset.

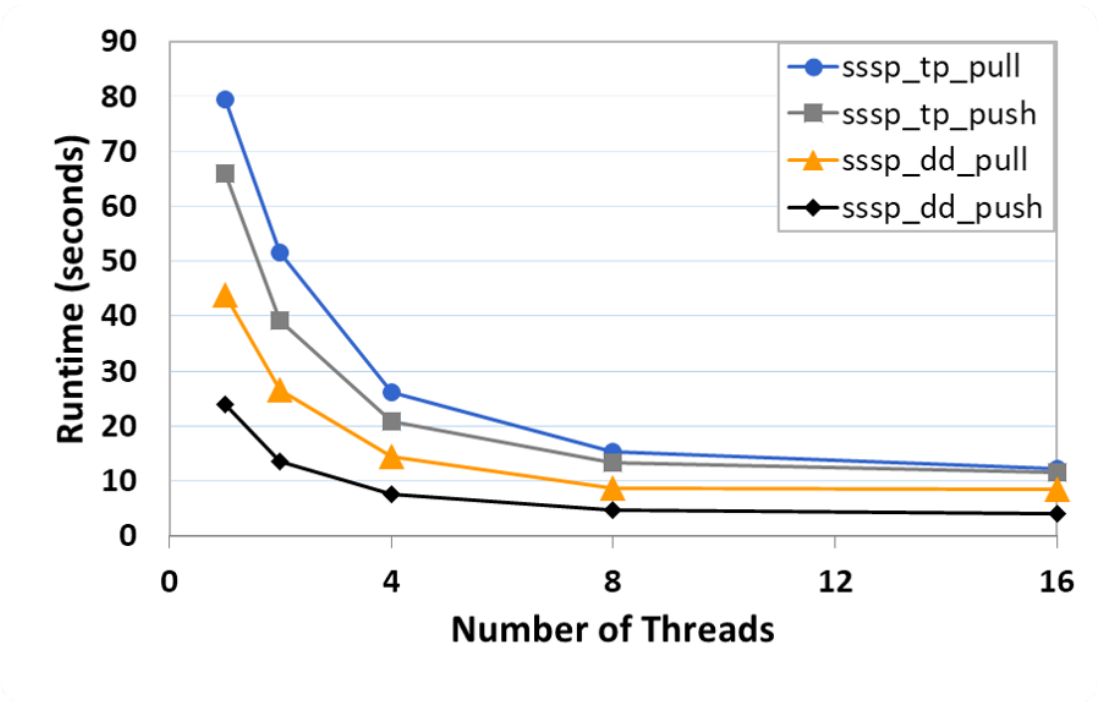

Figure 5.10: Execution time of SSSP with *rmat* dataset.

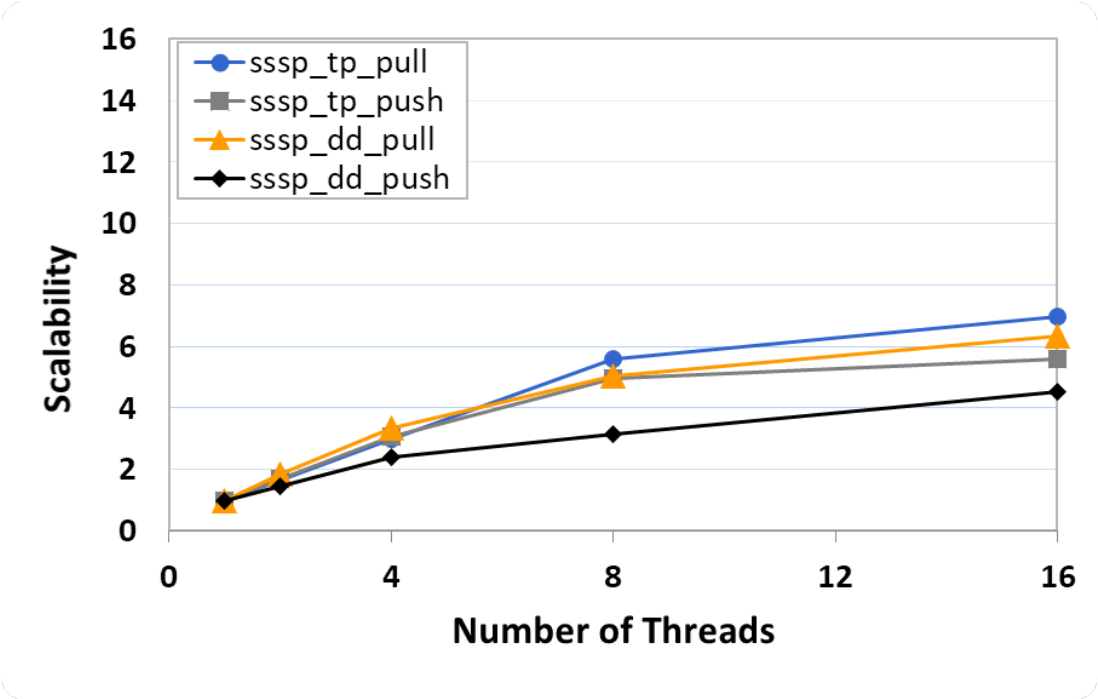

Figure 5.11: Scalability of SSSP with *pld* dataset.

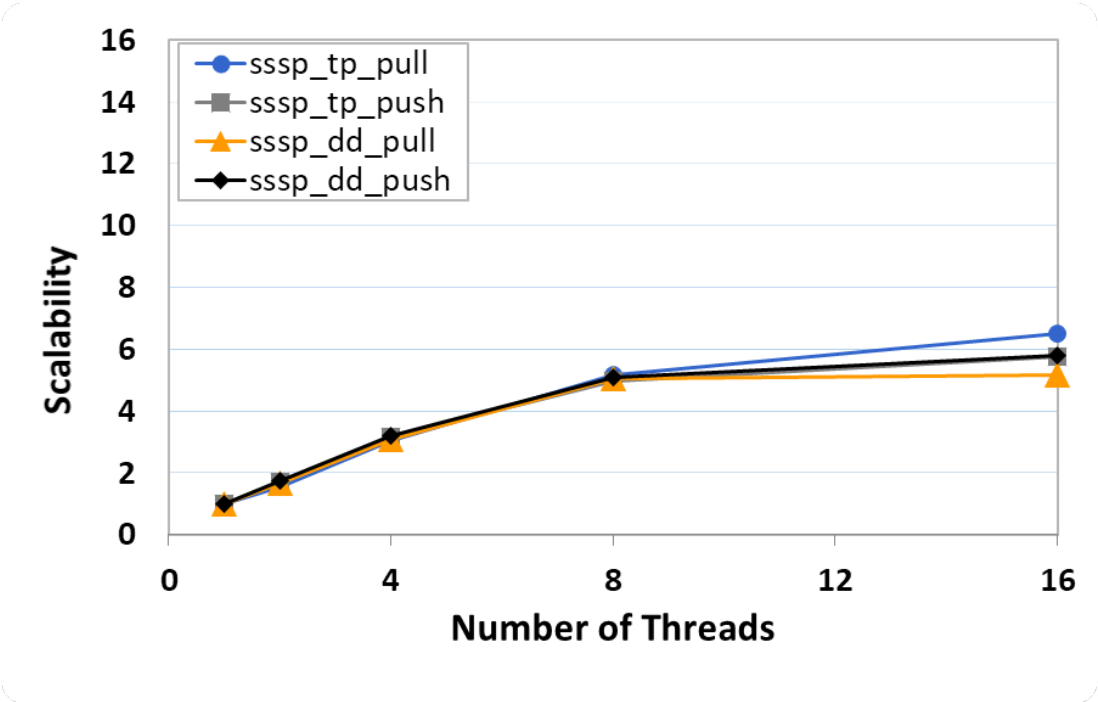

Figure 5.12: Scalability of SSSP with *rmat* dataset.

5.2.1.3 BFS

BFS implementations exhibit similar performance when compared to SSSP. For both classes, data-driven methods are faster than topology-driven methods, and push-based alternatives outperform pull-based alternatives as illustrated in Figures 5.13 and 5.14. Moreover, Figures 5.15 and 5.16 show that scalability of data-driven methods is lower than the scalability of topology-driven ones. The increase in the number of threads improves the performance of all algorithms. One interesting observation which can be seen from Figure 5.16 is that the effect of data access patterns such as pull vs. push on scalability of the algorithms on *rmat* graph is negligible.

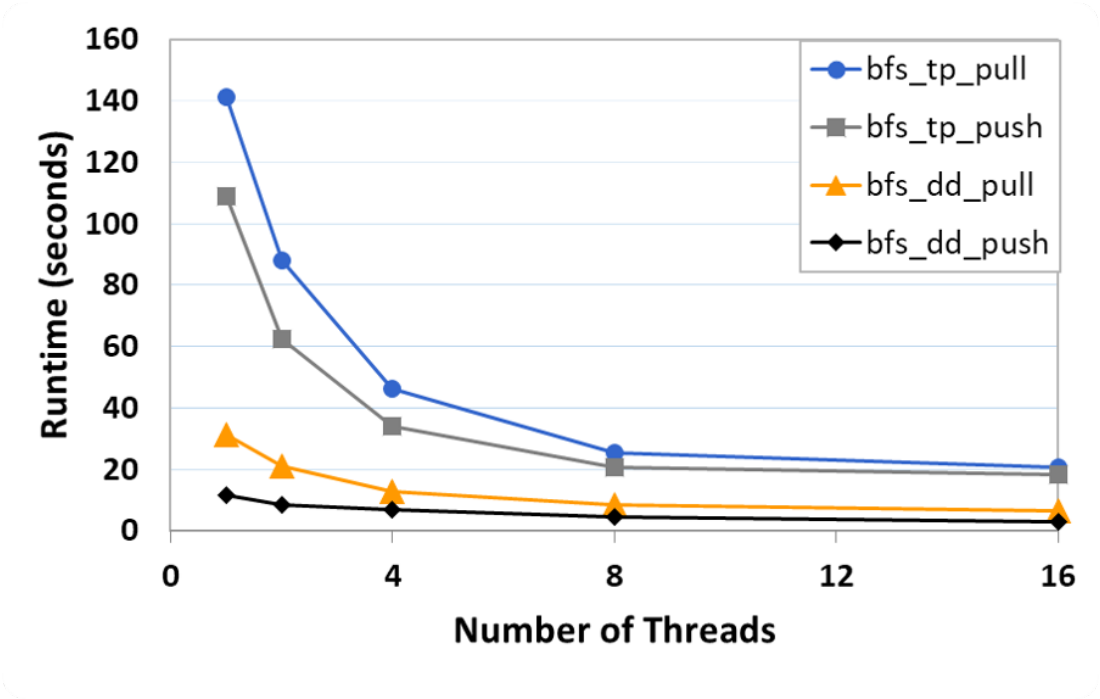

Figure 5.13: Execution time of BFS with *pld* dataset.

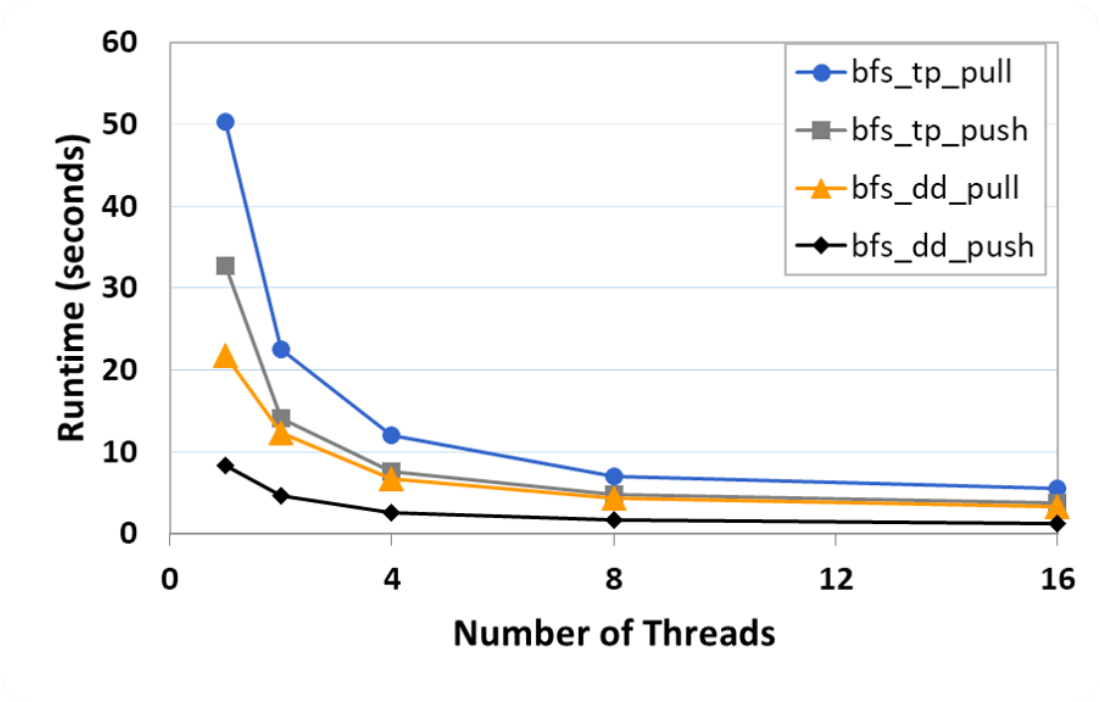

Figure 5.14: Execution time of BFS with *rmat* dataset.

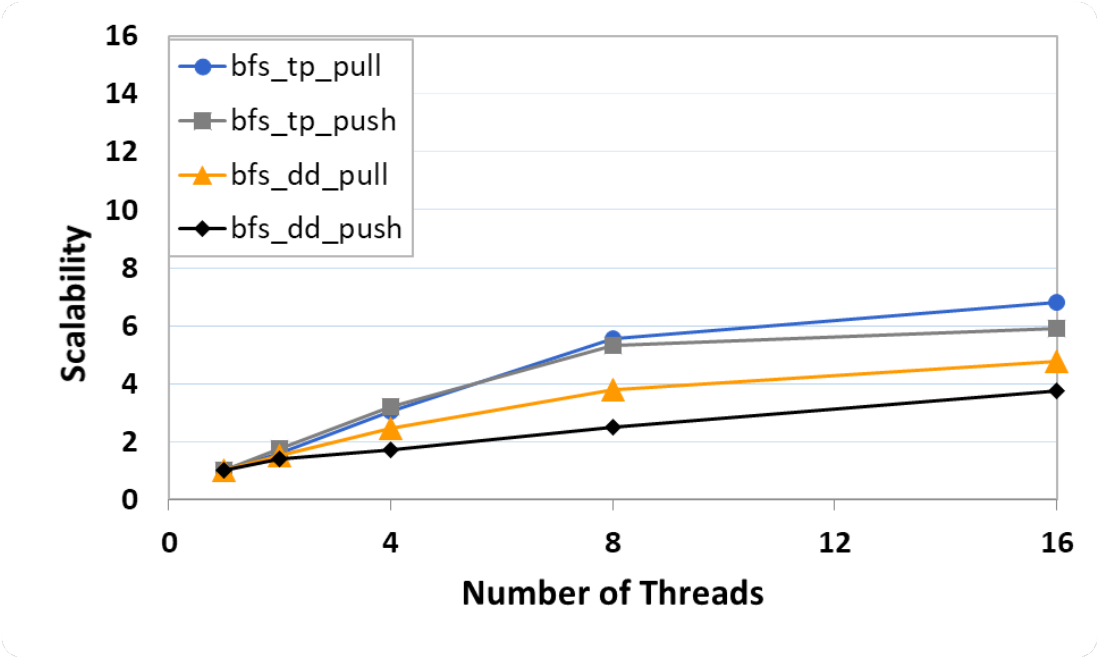

Figure 5.15: Scalability of BFS with *pld* dataset.

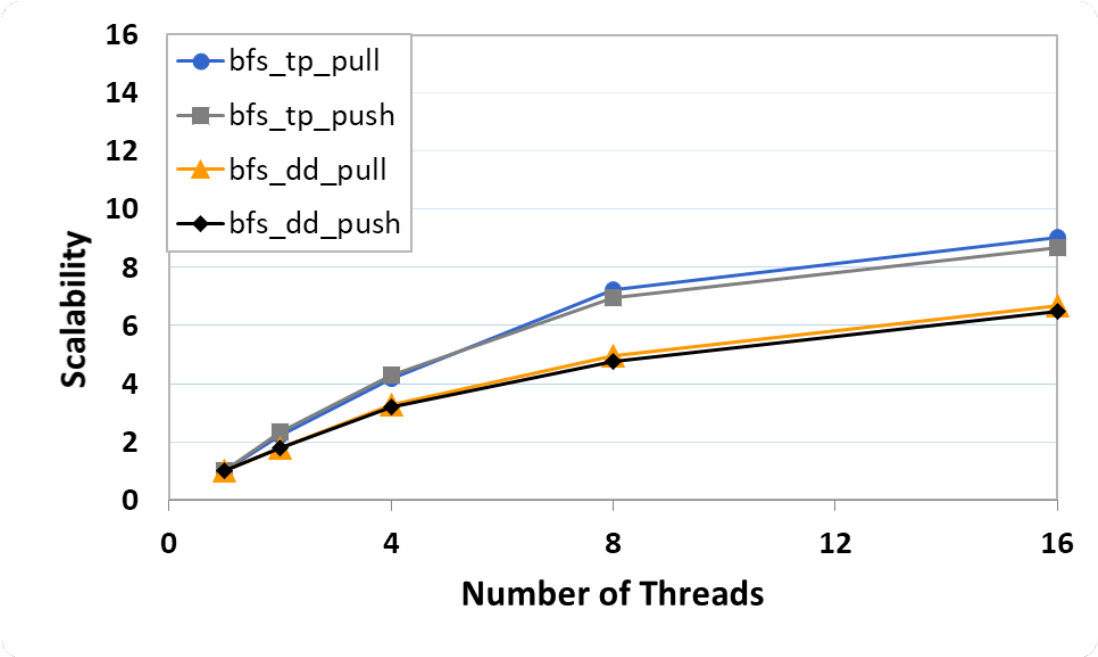

Figure 5.16: Scalability of BFS with *rmat* dataset.

5.2.2 Speedups

In this section, we report speedups for the aforementioned implementations of PR, SSSP, and BFS. For brevity, we only report speedups on large graphs which are still valid for small graphs. We normalize the execution time speedups to the synchronous topology-driven pull-based method since it is a widely used benchmark in previous studies. For PR, the base case corresponds to the power method as shown in Equation 4.1.

As already mentioned in the previous chapter, data-driven push-based methods provide the highest performance in all the experiments. For PR algorithm, Figures 5.17 and 5.18 demonstrate that speedups of work efficient push based implementations are 35x and 11x for *pld* and *rmat* graphs, respectively. The least speedups are gained from topology-driven pull-push-based methods. Nevertheless, if pull-push based methods are implemented in a data-driven way, they deliver substantial speedups over their topology-driven versions. For instance, on *pld* graph, the speedup of pull-push-based implementation increases from 5x to 18x when a worklist is used in order to track previously visited nodes.

As can be seen in Figures 5.19-20 and 5.21-22, all implementations of SSSP and BFS algorithms show at least 7x speedup relative to the baseline implementation in 16 threads. For SSSP and BFS applications, significantly high speedups are obtained in *pld* and *rmat* graphs when the algorithms utilize a worklist and do computations in the push direction. Figures 5.19 and 5.21 illustrate that for *pld* graph, speedups of SSSP and BFS algorithms in 16 threads are 38x and 46x, respectively. Similarly, as shown in Figures 5.20 and 5.22, for *rmat* graph, speedups of SSSP and BFS algorithms in 16 threads are 19x and 40x, respectively. Although BFS algorithm is implemented in a data-driven push-based way, it delivers substantial speedups for both datasets when compared to other implementations. On the other hand, all classes of data-driven implementations of SSSP algorithm delivers higher speedups than all topology-based versions. We also notice that data access patterns do not have a huge impact on speedups of SSSP and BFS when the algorithm activates each node in every iteration.

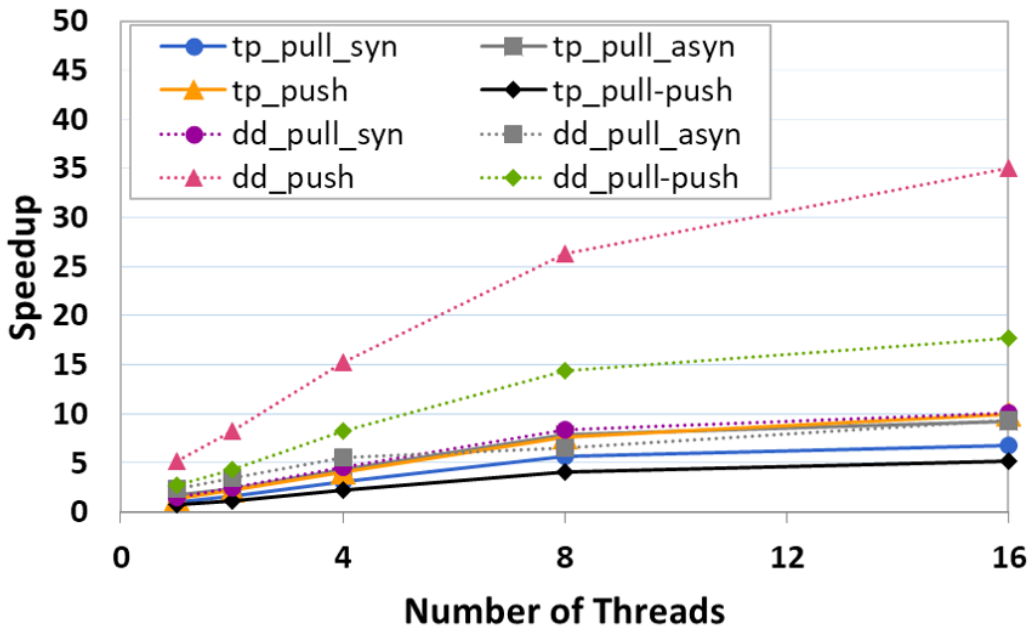

Figure 5.17: Speedups observed for PR with *pld* dataset.

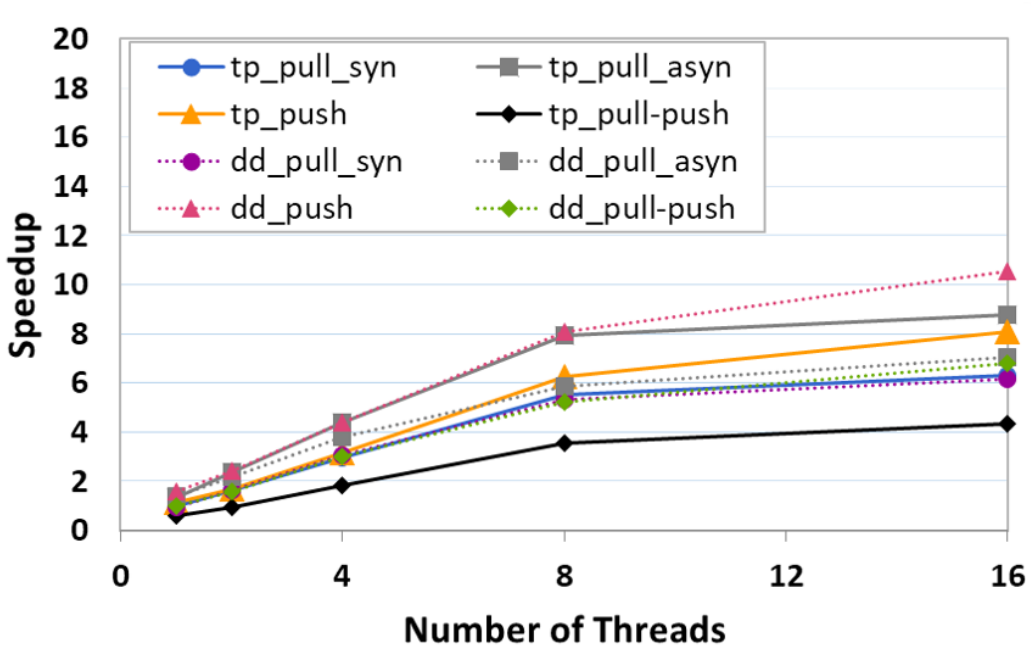

Figure 5.18: Speedups observed for PR with *rmat* dataset.

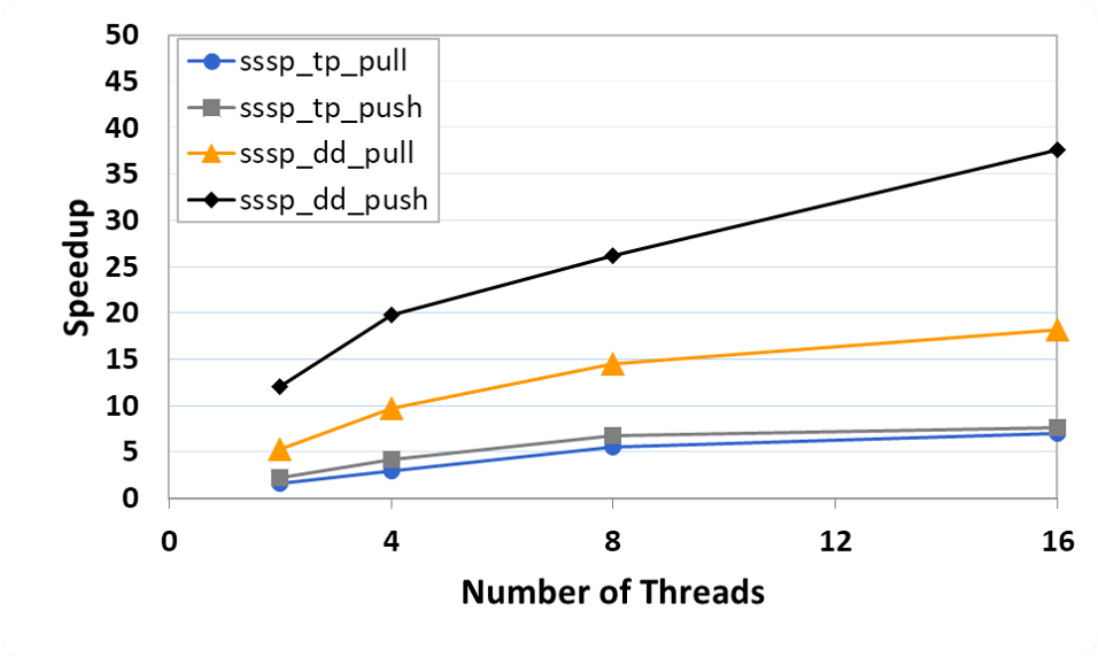

Figure 5.19: Speedups observed for SSSP with *pld* dataset.

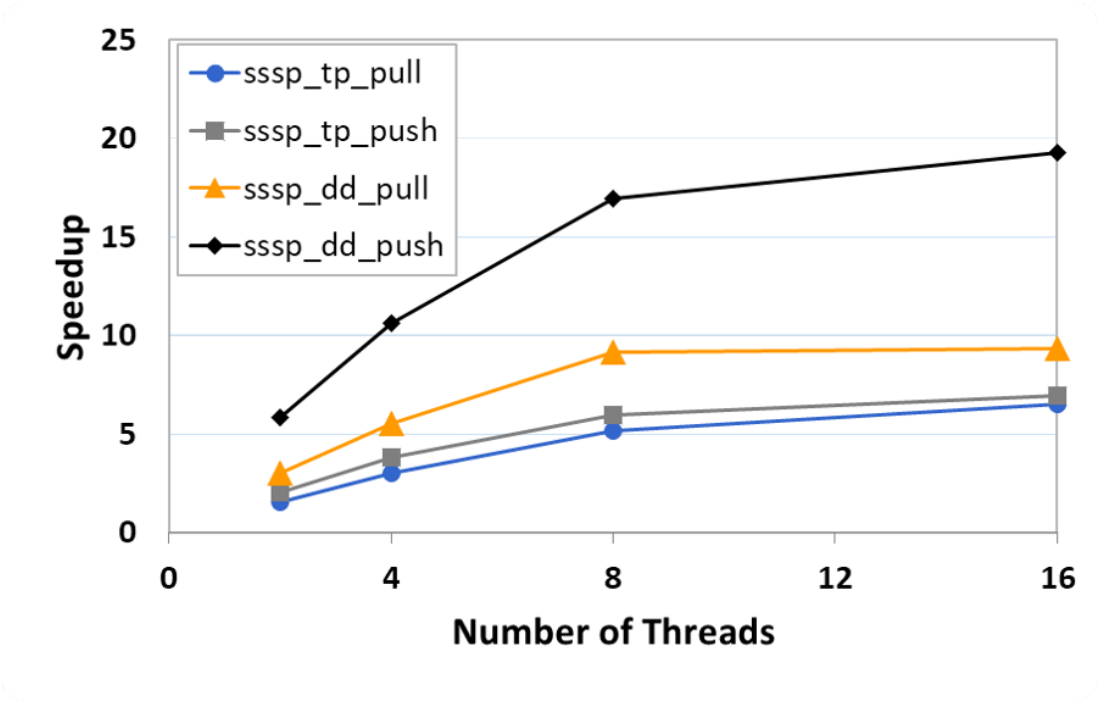

Figure 5.20: Speedups observed for SSSP with *rmat* dataset.

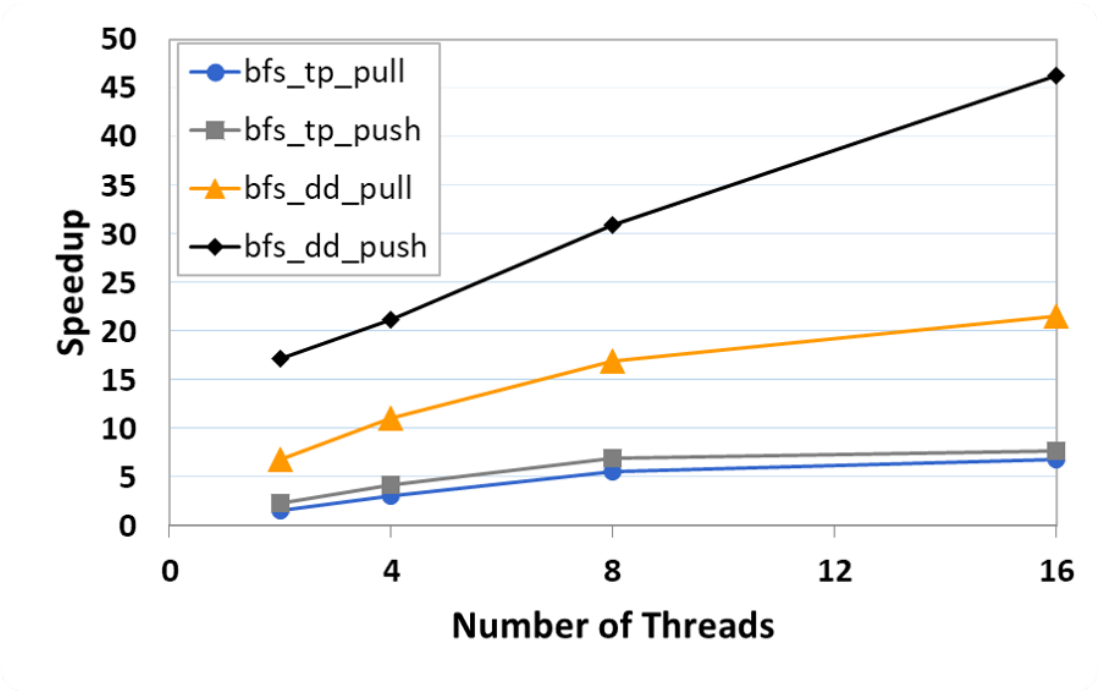

Figure 5.21: Speedups observed for BFS with *pld* dataset.

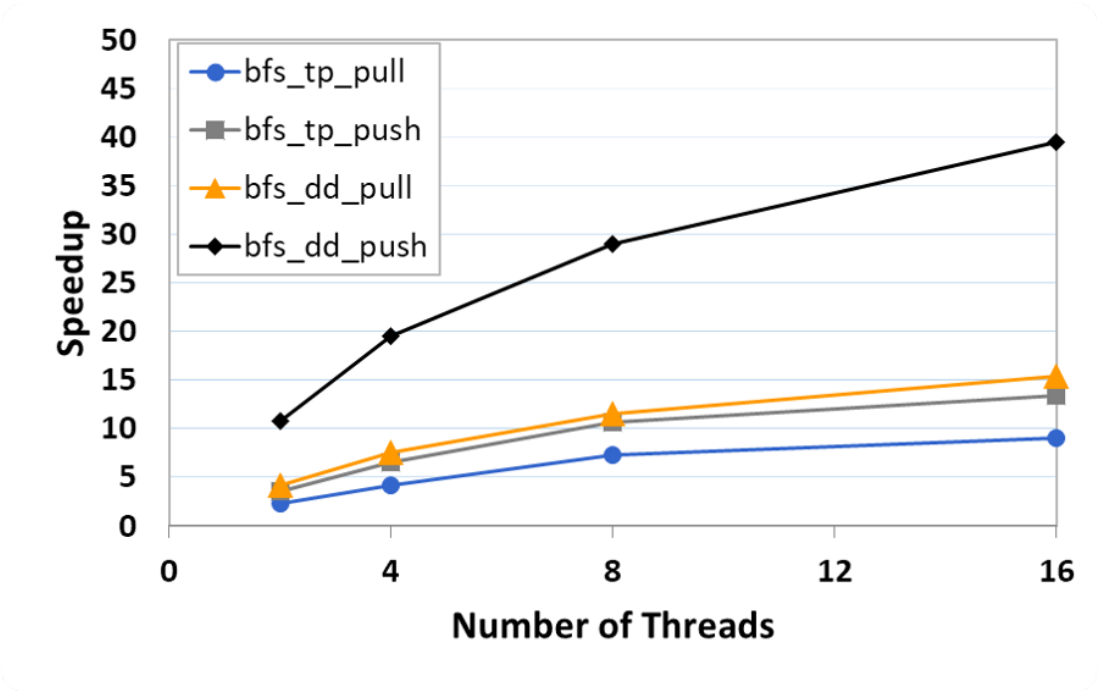

Figure 5.22: Speedups observed for BFS with *rmat* dataset.

5.2.3 Work Efficiency

In Section 5.2.1, we observed that the performance of data-driven methods is better than the performance of topology driven ones. Also, the authors showed when asymmetric convergence is enabled in PR algorithm, the total number of edges processed is reduced by 47 percent on average [47]. Therefore, work efficiency can be an important metric for analyzing the performance of graph applications. For these reasons, we provide two different metrics in order to calculate work efficieny. First, we keep a track of edges processed since it captures the number of memory accesses and multiply-add operations applied on edges. The second metric is the number of nodes activated during the execution of an application. For topology-driven applications, since the number of edges processed per step and nodes activated per step are constant, total work done is calculated by multiplying the number of edges/nodes of an input graph with the number of iterations.

Our first observation is that data-driven push-based application is the most work efficient implementation among all applications as shown in Figures 5.23 and 5.24. Similarly, we already seen that data-driven push-based application is the fastest one as shown in Figures 5.3 and 5.4. These two findings are consistent with each other since data-driven push-based application performs the least work, hence it shows the best performance.

As shown in Figures 5.23 and 5.24, when applications use a worklist in order to track active vertices, they reduce the number of edges processed significantly. Especially, data-driven applications of both SSSP and BFS reduce their total work done in a substantial amount relative to topology-driven pull-based applications. Similarly, as shown in Figures 5.25 and 5.27, data-driven applications implemented in the push and pull-push direction process 5.1x and 4.5x fewer edges on *pld* graph. As a result, they perform better than their topology-driven versions. This observation is important since when applications become work efficient, they can utilize cache better, thereby improving performance.

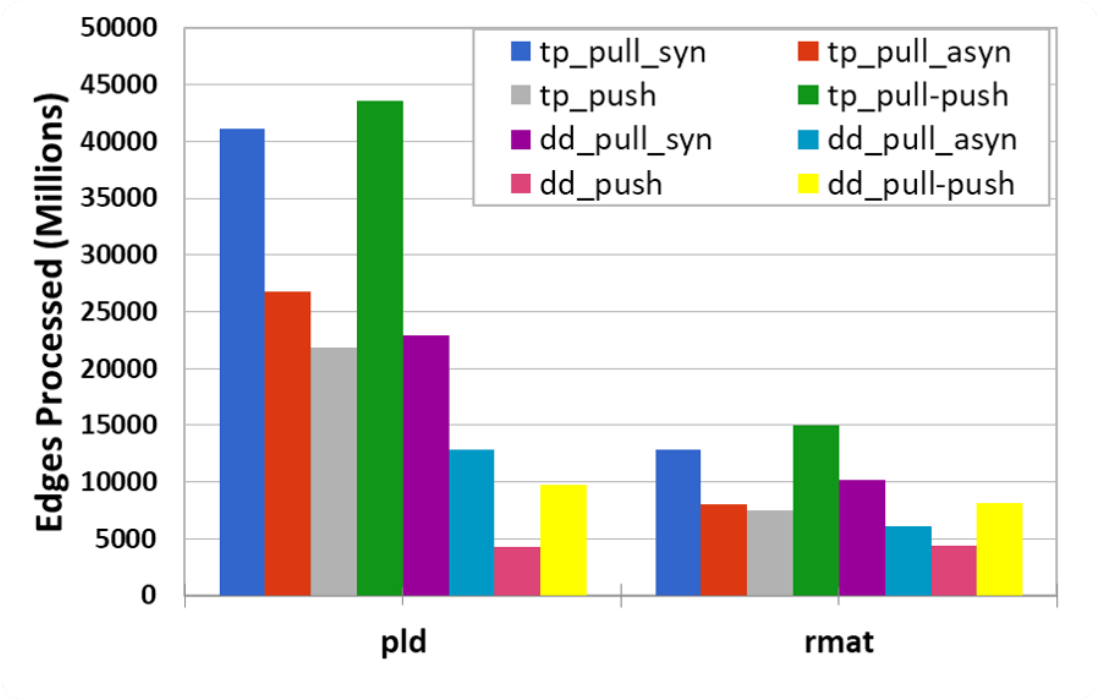

Figure 5.23: Total edges processed for PR with *pld* and *rmat* graphs.

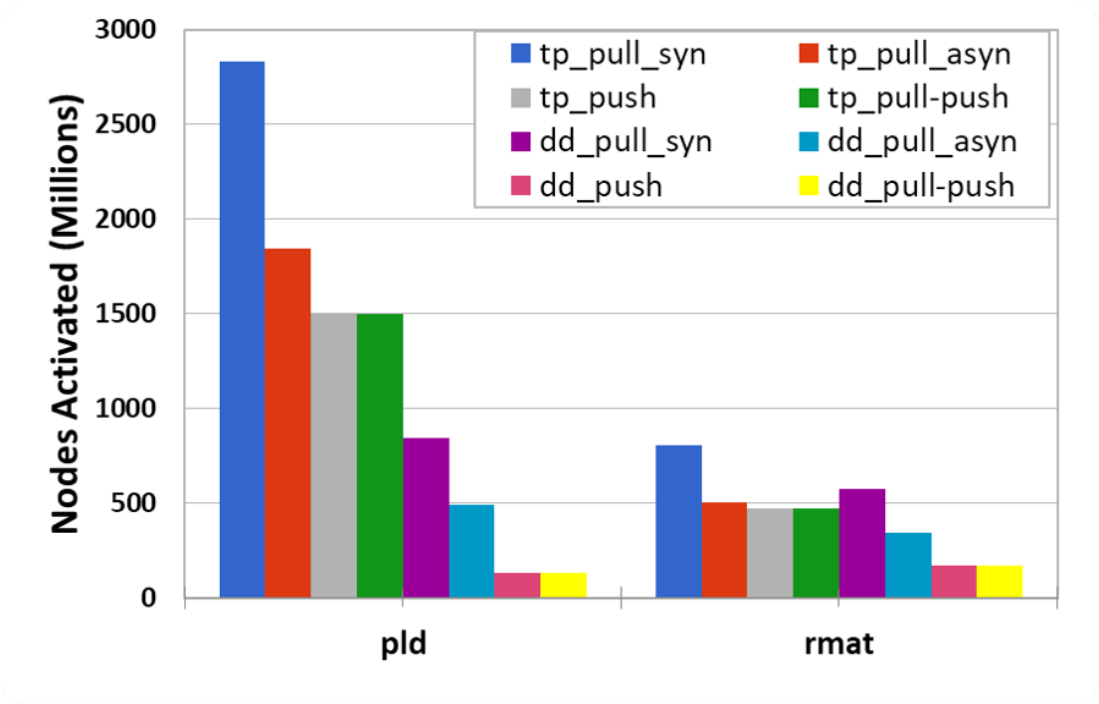

Figure 5.24: Total nodes activated for PR with *pld* and *rmat* graphs.

Data access patterns affect the number of edges processed when applications do not consider activation status of a node. As can be seen in Figure 5.24, both push-based and pull-push based methods activate the same number of nodes; however, Figure 5.23 shows that the pull-push based method processes more edges than the push-based method. Although pull-push based application speeds up the convergence rate of the algorithm like push-based method, it does extra work on incoming edges as shown in Figure 5.24. Since the number of edges processed changes the number of memory accesses, the increase in work done results in poor performance. Similarly, pull-push based methods process more edges in spite of activating less nodes than pull-based method as illustrated in Figures 5.23 and 5.24, respectively. Likewise, Figures 5.25 and 5.27 show that push-based applications of SSSP and BFS process less number of edges when compared to their pull-based counterparts. Additionaly, topology driven push-based versions of SSSP and BFS algorithms are very inefficient in terms of total nodes activated.

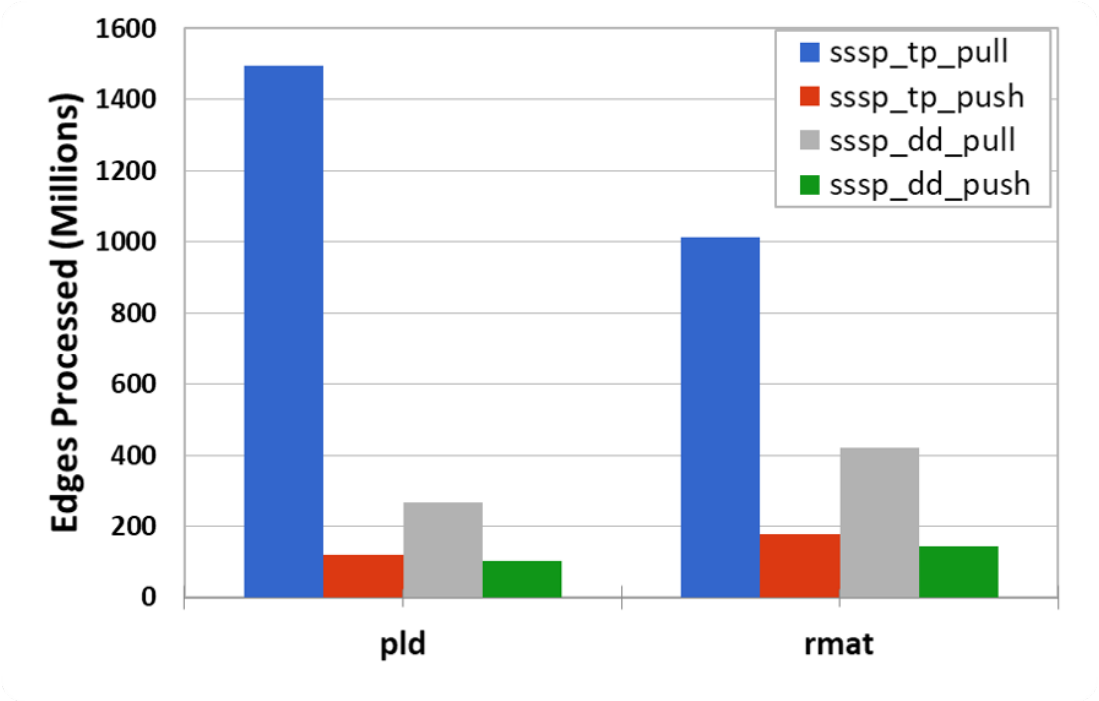

Figure 5.25: Total edges processed for SSSP with *pld* and *rmat* graphs.

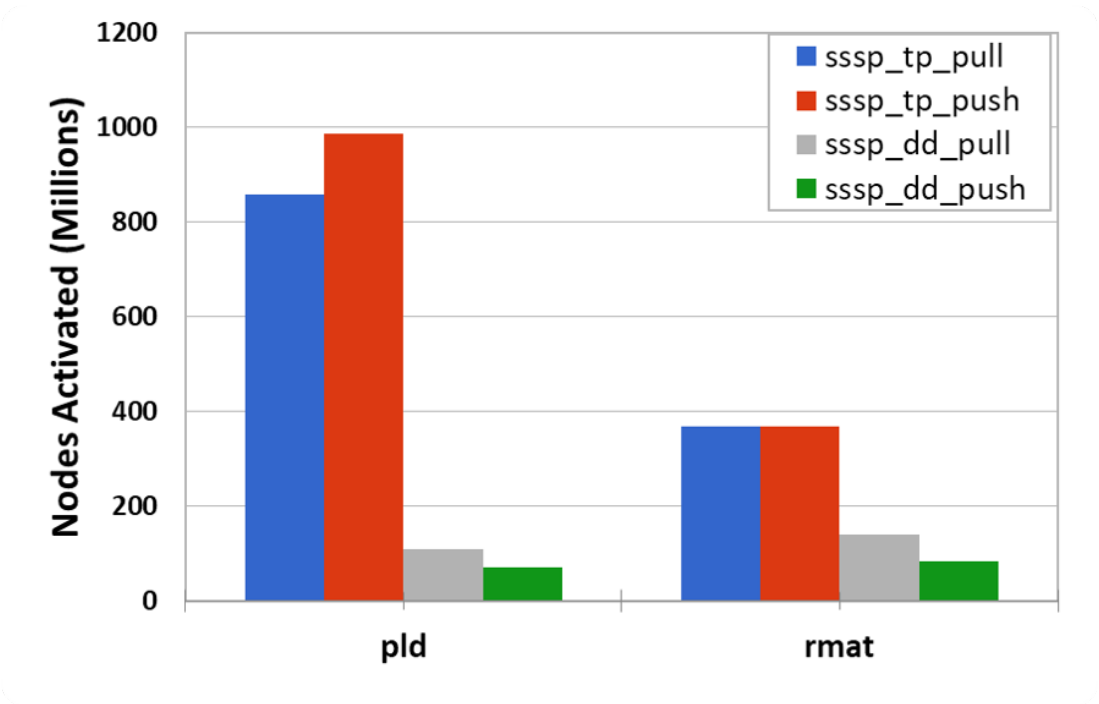

Figure 5.26: Total nodes activated for SSSP with *pld* and *rmat* graphs.

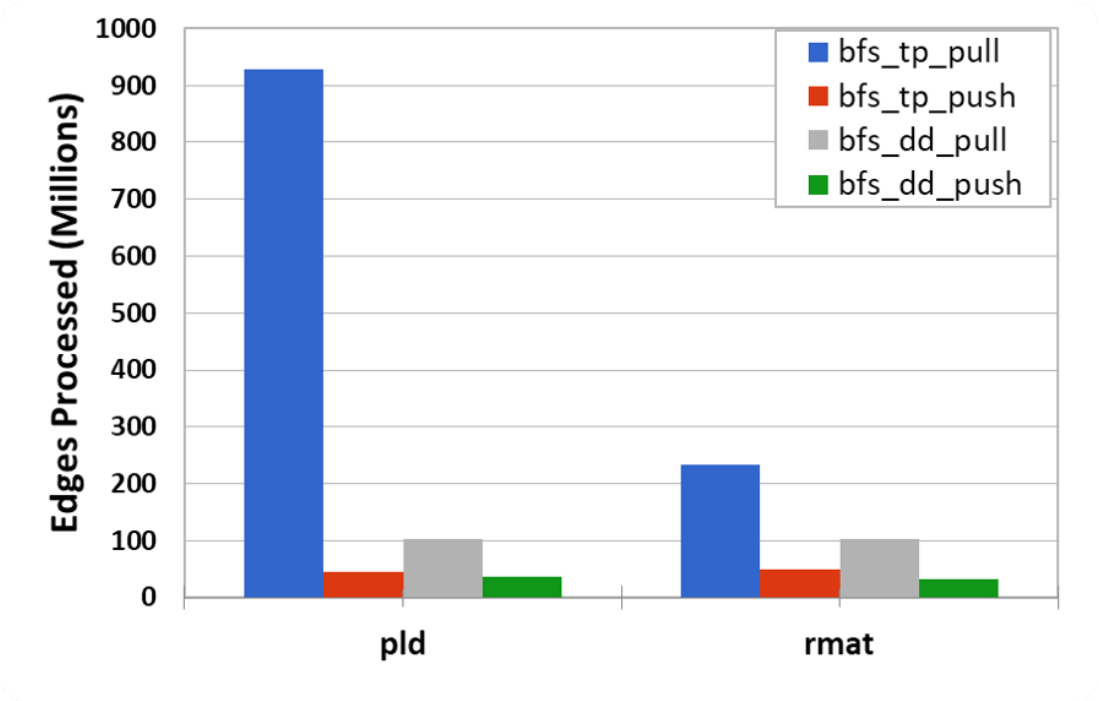

Figure 5.27: Total edges processed for BFS with *pld* and graphs.

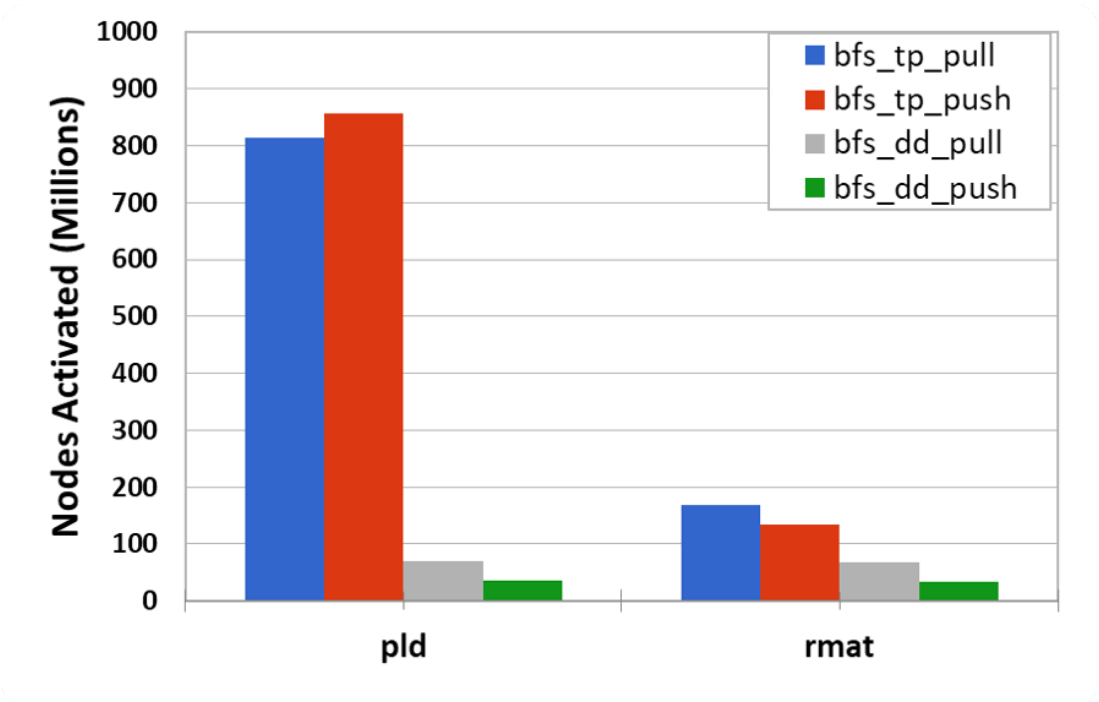

Figure 5.28: Total nodes activated for BFS with *pld* and *rmat* graphs.

5.3 Microarchitectural Results

In this section, we discuss cache behavior of the aforementioned applications. In addition, we present the rate of translation lookaside buffer (TLB) miss rates and the average number of instructions executed per cycle (IPC) for each application. We evaluate the performance of the system by using hardware performance counters.

5.3.1 L2/L3 Miss Rates

Although there are many factors to consider when determining performance, it is not clear which application shows better performance. For this purpose, we analyze cache behavior for different implementations explained earlier.

We implement pull, push, and pull-push variants of PR algorithm for each work activation model. Although each application shows different performance, the variations between their L3 miss rates is negligible. We observe that type of an application does not affect the L3 significantly but the graph size for the application being used is directly proportional to the number of L3 misses. For instance, L3 miss rates are less than 55% on small graphs, whereas they are higher than 75% on large graphs, as shown in Figure 5.30.

For all applications, we obtain significantly higher L2 miss rates which are beyond 80%. In spite of low hit rates, we observe that push-based implementations result in lower L2 hit rates on all graphs, as shown in Figure 5.29. In addition, *lj* graph delivers lower L2 miss rates for all applications except push-based versions when compared to the other input graphs.

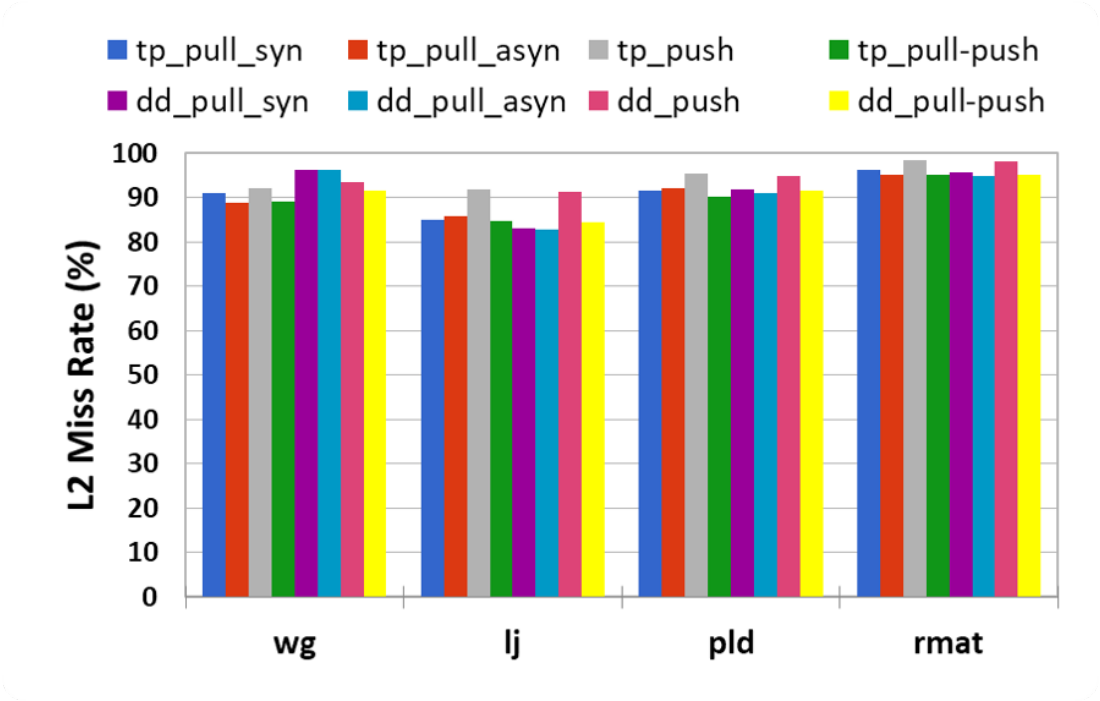

Figure 5.29: L2 miss rates for PR with *wg*, *lj*, *pld*, and *rmat* graphs.

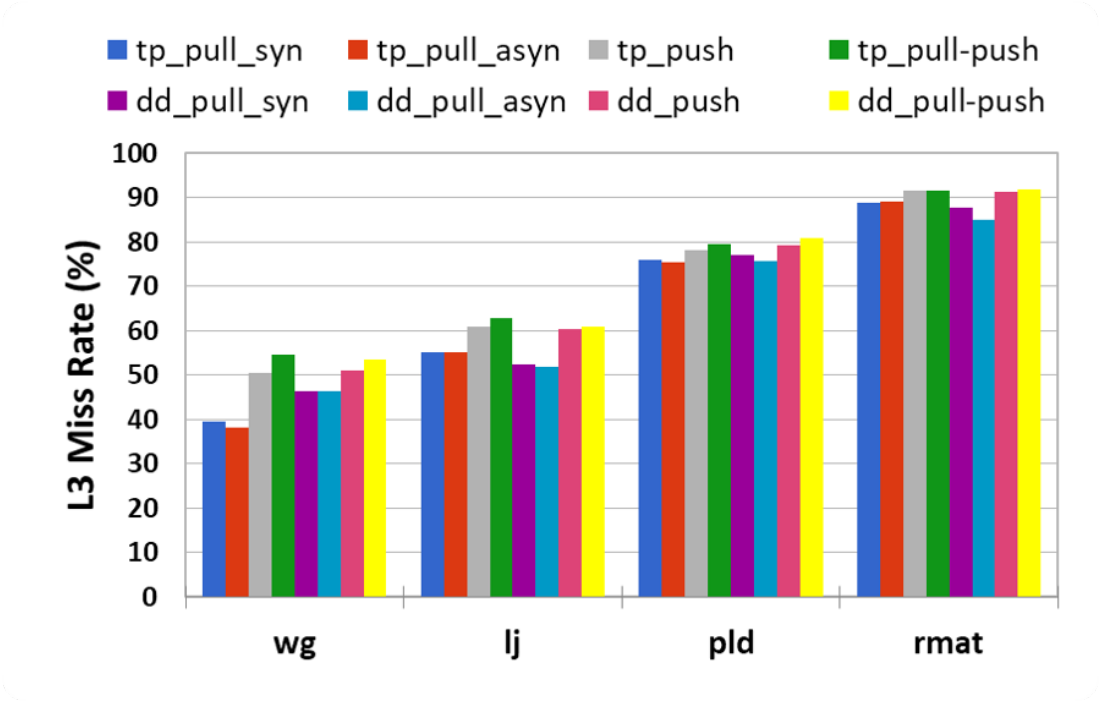

Figure 5.30: L3 miss rates for PR with *wg*, *lj*, *pld*, and *rmat* graphs.

Figures 5.32 and 5.34 demonstrate that SSSP and BFS applications show a similar pattern with PR application when L3 miss rates are considered. When the size of input graph increases, the total miss rate increases. However, for small graphs, it does not go over 50%. Likewise, all variants of SSSP and BFS generate high miss rates like PR. The highest hit rates for all applications are obtainted with *lj* graph among all input graphs. However, same observations made for PR are still valid for SSSP and BFS applications as shown in Figures 5.31-32 and Figures 5.33-34, respectively.

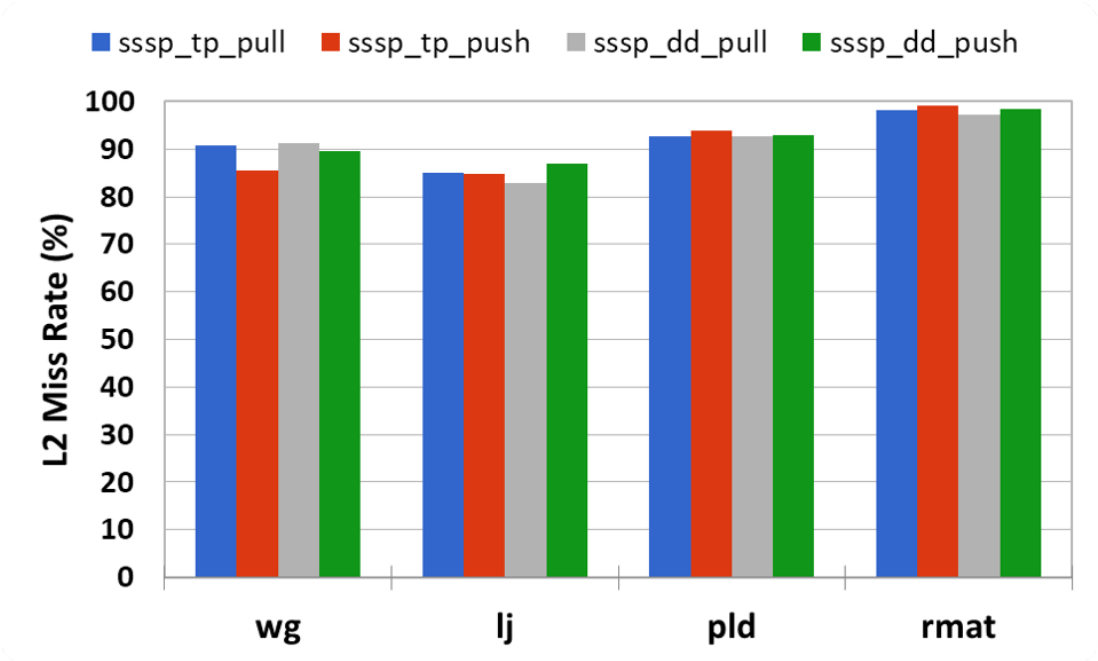

Figure 5.31: L2 miss rates for SSSP with *wg*, *lj*, *pld*, and *rmat* graphs.

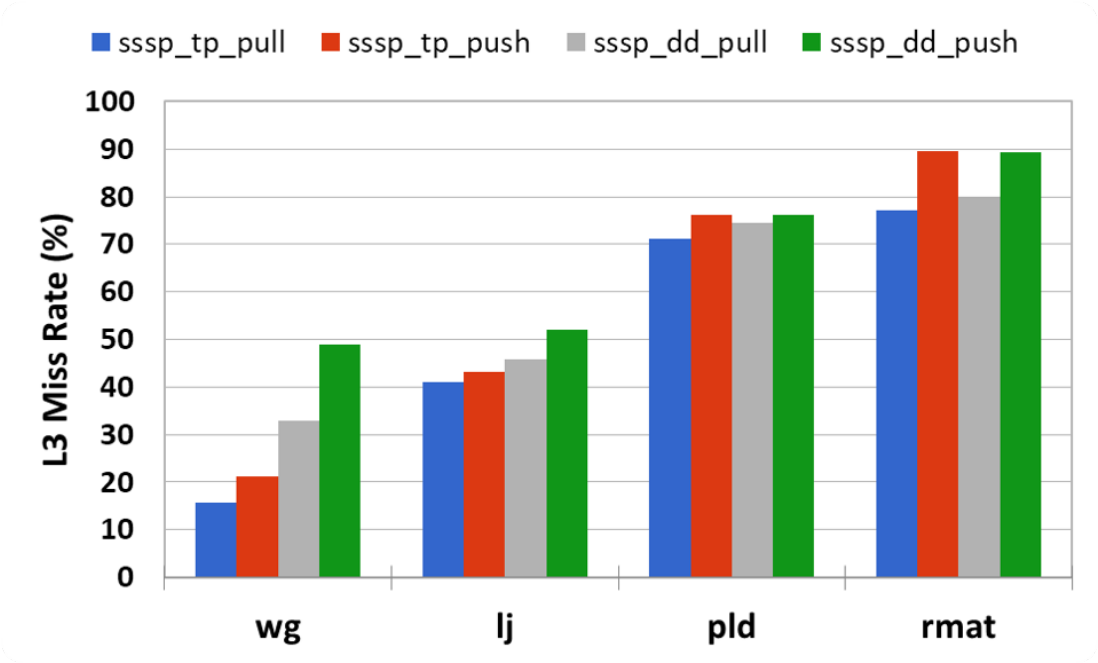

Figure 5.32: L3 miss rates for SSSP with *wg*, *lj*, *pld*, and *rmat* graphs.

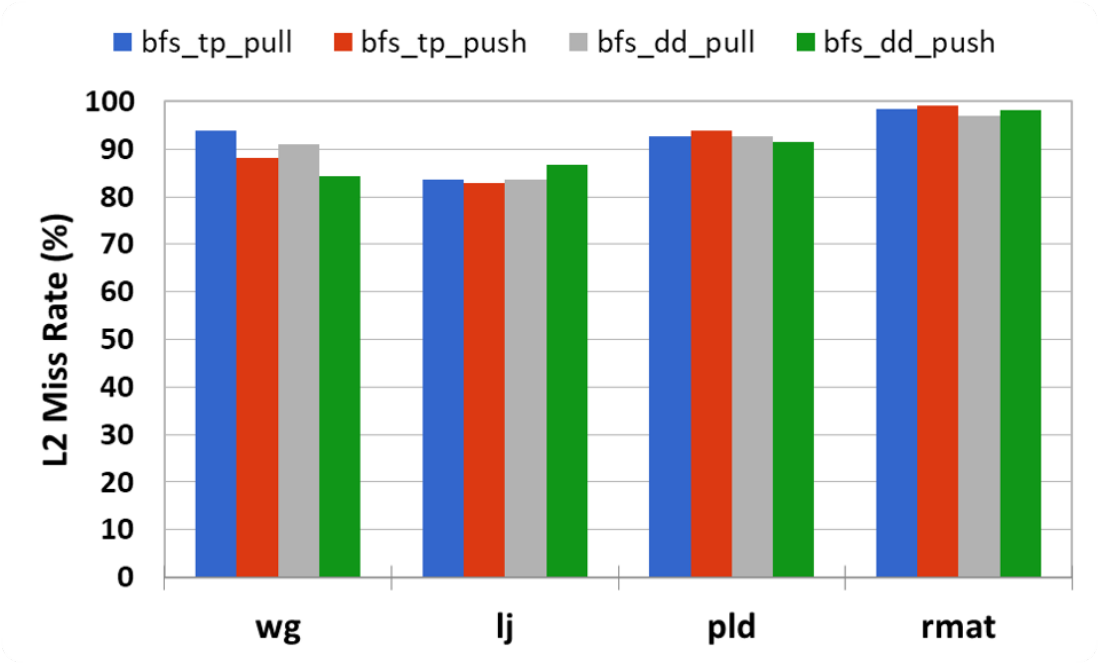

Figure 5.33: L2 miss rates for BFS with *wg*, *lj*, *pld*, and *rmat* graphs.

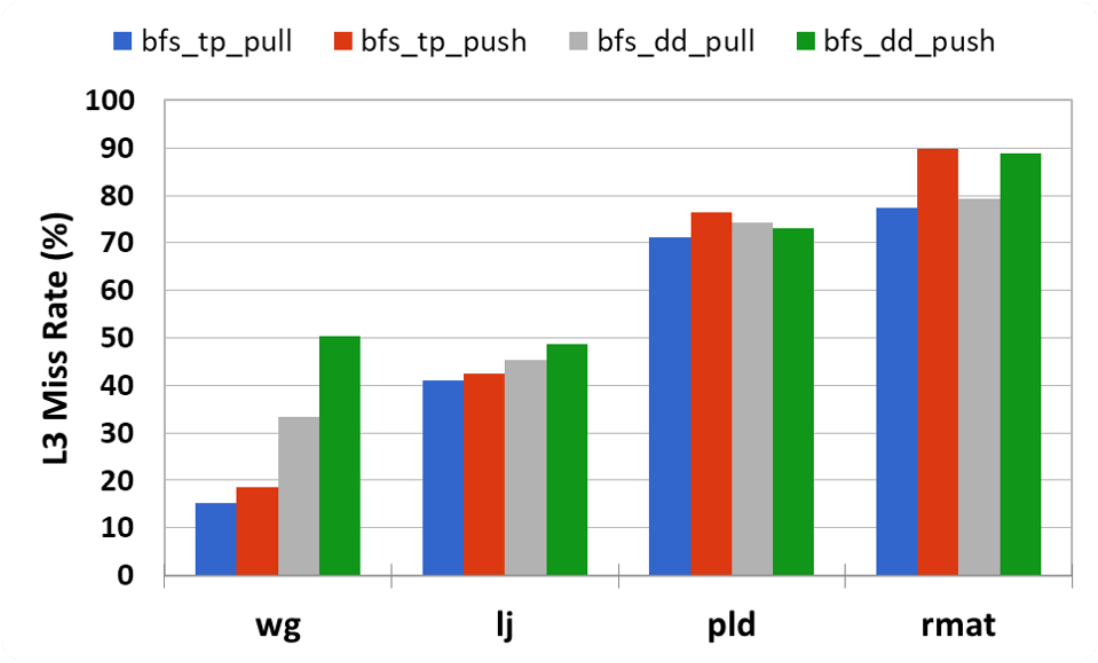

Figure 5.34: L3 miss rates for BFS with *wg*, *lj*, *pld*, and *rmat* graphs.

5.3.2 L1/TLB Miss Rates

For PR algorithm, we notice that pull-based implementations show high L1 miss rates when compared to push and pull-push based variants, as shown in Figure 5.35. Especially, L1 hit rates become worse when a pull-based algorithm is executed synchronously.

Work activation does not make much difference for different application counterparts except asynchronous pull-based applications. For instance, L1 miss rates for both synchronous pull, push, and pull-push based applications are slightly different from each other. However, L1 hit rates decrease when an asynchronous pull-based algorithm is implemented by utilizing a worklist structure.

We expect pull-based applications to be cache friendly since they perform frequent read operations on in-edges and only perform one update operation. However, our findings show the opposite of this expectation. Push-based applications achieve higher hit rates when compared to their pull-based versions. Furthermore, all variants of applications show low L1 and TLB miss rates. One reason for the decrease in miss rates when compared to L2/L3 miss rates is the regular accesses thanks to edge lists. For each application, we keep two separate lists for in-edges and out-edges per vertex so applications can benefit from increased spatial locality.

As can be seen from Figure 5.36, patterns observed for L1 miss rates for each data access modes are also valid for TLB miss rates. Moreover, TLB miss rates increase with dataset sizes, as expected, since the bigger the data the higher the number of pages to store. This situation puts more pressure on TLB entries, and as a result, TLB entries are evicted more frequently.

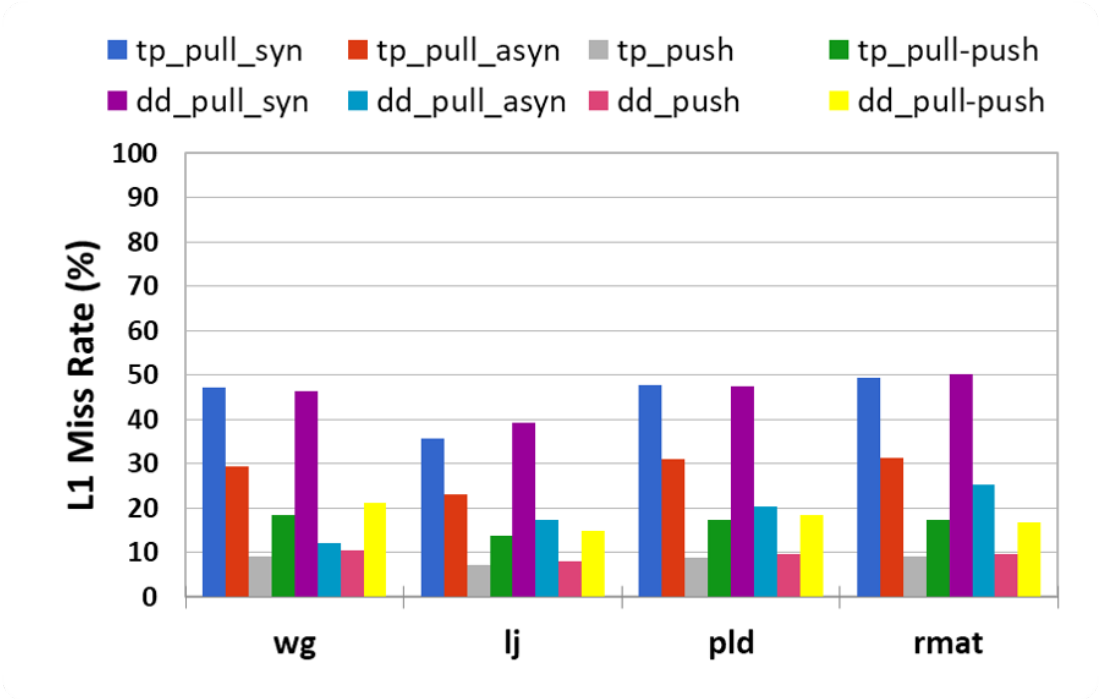

Figure 5.35: L1 miss rates for PR with *wg*, *lj*, *pld*, and *rmat* graphs.

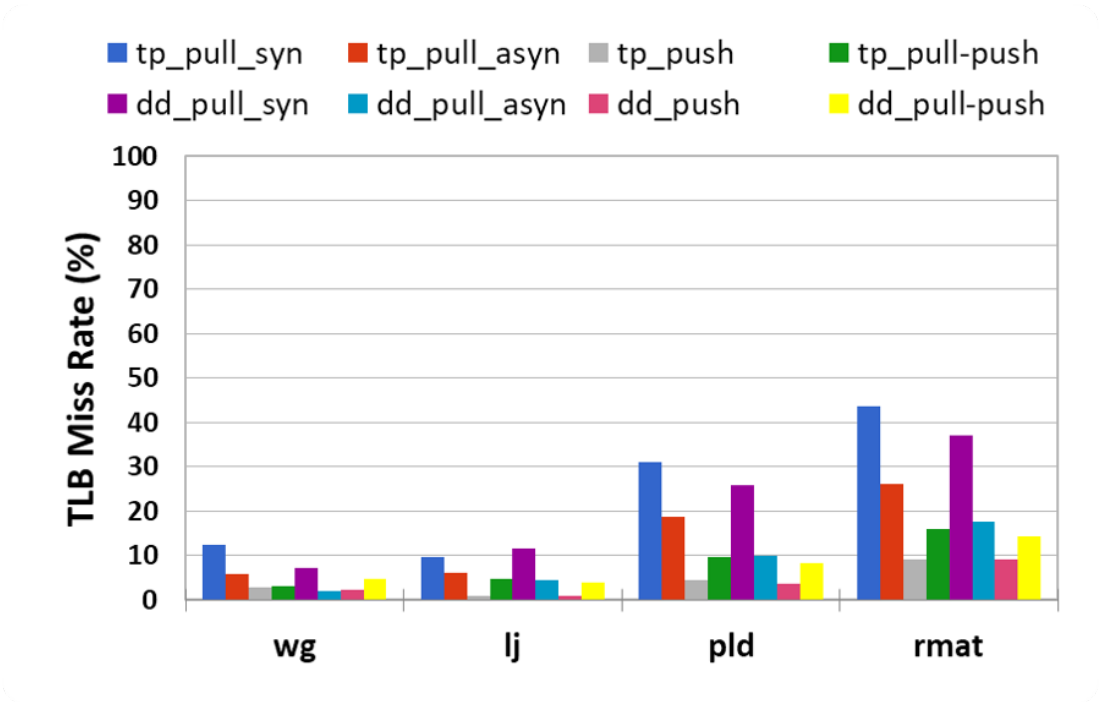

Figure 5.36: TLB miss rates for PR with *wg*, *lj*, *pld*, and *rmat* graphs.

Similar to PR, SSPP and BFS show similar behavior on L1 and TLB miss rates, as can be seen in Figures 5.37 and 5.38 for SSSP and in Figures 5.39 and 5.40 for BFS. Total L1 and TLB miss rates improve when compared to L2/L3 miss rates. Secondly, push-based applications have higher L1 and TLB hit rates in contrast to pull-based applications. Moreover, similar to PR, TLB miss rates improve on small graphs, whereas they worsen on large graphs.

Finally, L1 and TLB rates have same patterns for each data access pattern. Moreover, we observed that *wg* and *pld* graphs are more responsive than *lj* and *rmat* graphs to work activation. As shown in Figures 5.38 and 5.40, when an application is designed in a data-driven way, both pull-based and push-based implementations on *wg* and *pld* graphs improve their L1 and TLB miss rates.

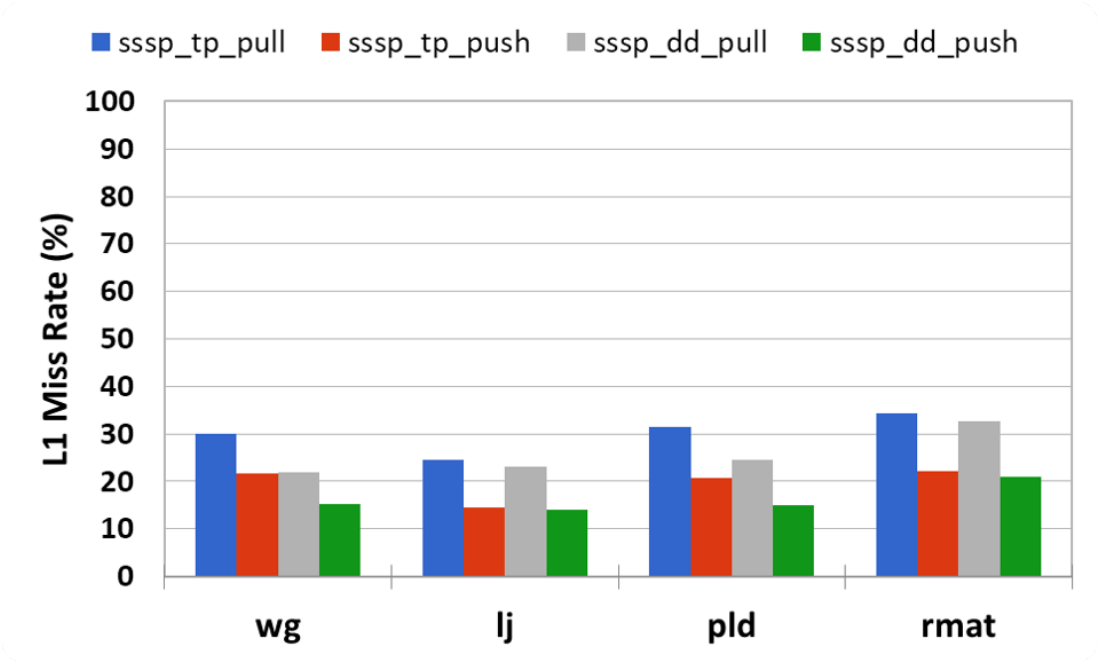

Figure 5.37: L1 miss rates for SSSP with *wg*, *lj*, *pld*, and *rmat* graphs.

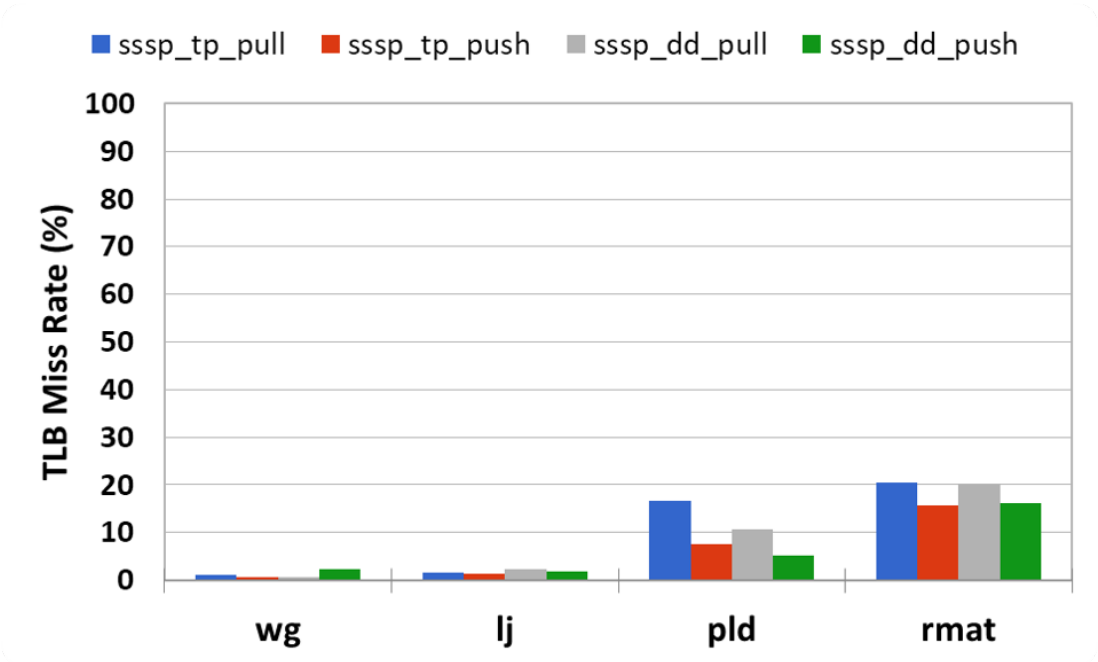

Figure 5.38: TLB miss rates for SSSP with *wg*, *lj*, *pld*, and *rmat* graphs.

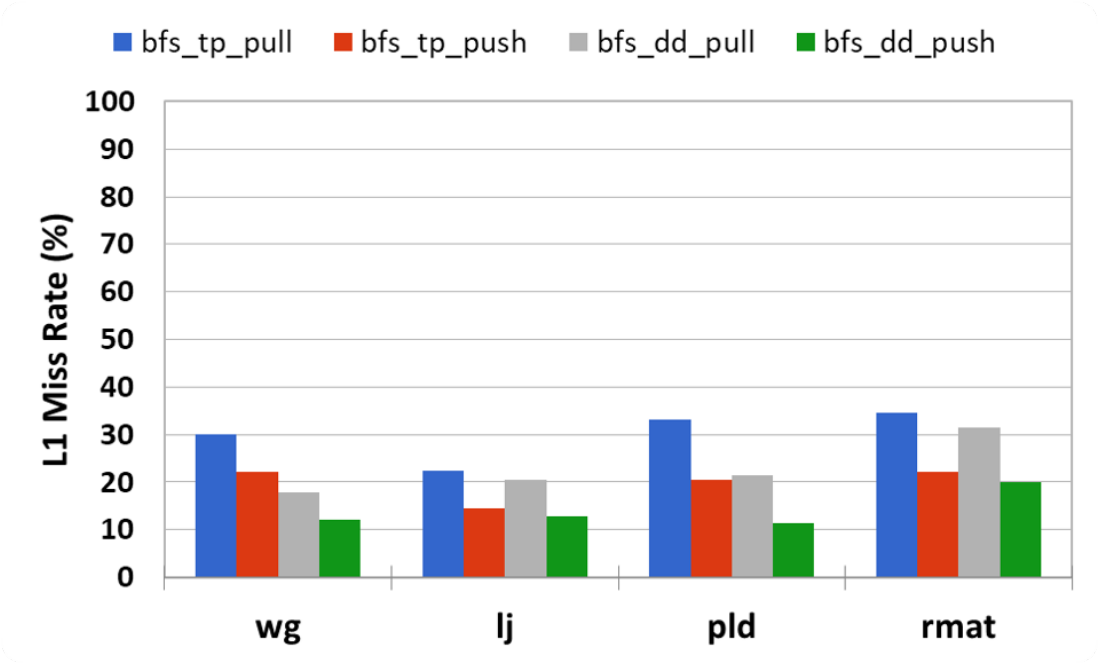

Figure 5.39: L1 miss rates for BFS with *wg*, *lj*, *pld*, and *rmat* graphs.

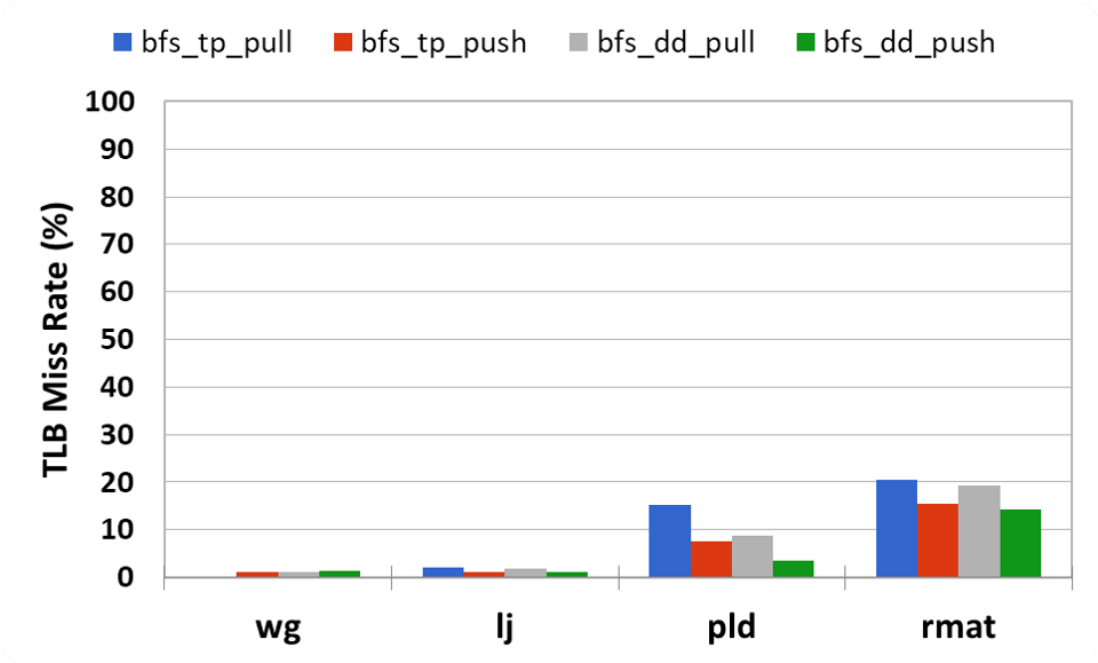

Figure 5.40: TLB miss rates for BFS with *wg*, *lj*, *pld*, and *rmat* graphs.

5.3.3 Instructions Per Cycle (IPC)

We also measure the number of instructions that each application execute per clock cycle. For PR applications, we observe a similar pattern on each input graph as shown in Figure 5.41. On the other hand, Figures 5.42 and 5.43 illustrate that SSSP and BFS applications show different behaviors with regards to the graph size. For instance, for all experiments except SSSP and BFS applications applied on small graphs, the overall IPC rates do not exceed 0.2. However, all variants of SSSP and BFS applications have an IPC value of more than 0.2, on small datasets.

Asynchronous data-driven pull-based method obtains the least IPC among all applications of PR as shown in Figure 5.41. By contrast, IPC values decrease for SSSP and BFS methods when they are data-driven as illustrated in Figures 5.42 and 5.43, respectively. Although applications have lower IPC on *rmat* graph compared to *pld*, they perform better for all implementations. In other words, maximizing IPC values does not improve performance. More specifically, best IPC value gives the worst results. For instance, for BFS and SSSP, IPC values are inversely proportional to execution times on small graphs as shown in Figures 5.42 and 5.43.

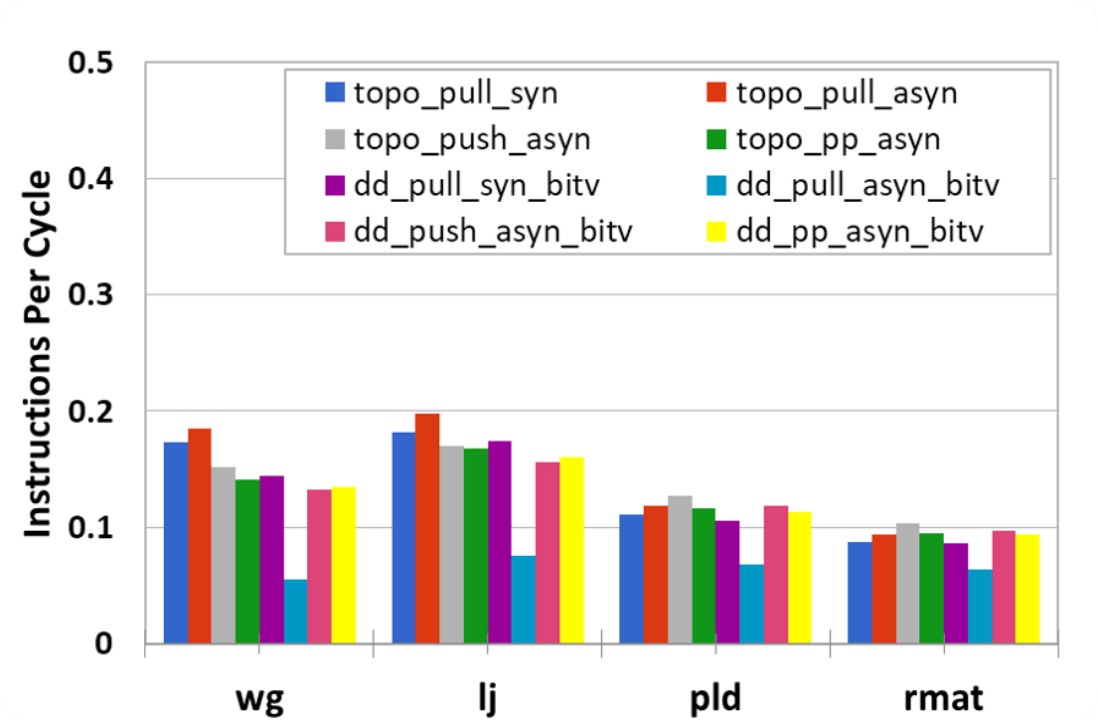

Figure 5.41: Instructions per cycle for PR with *wg*, *lj*, *pld*, and *rmat* graphs.

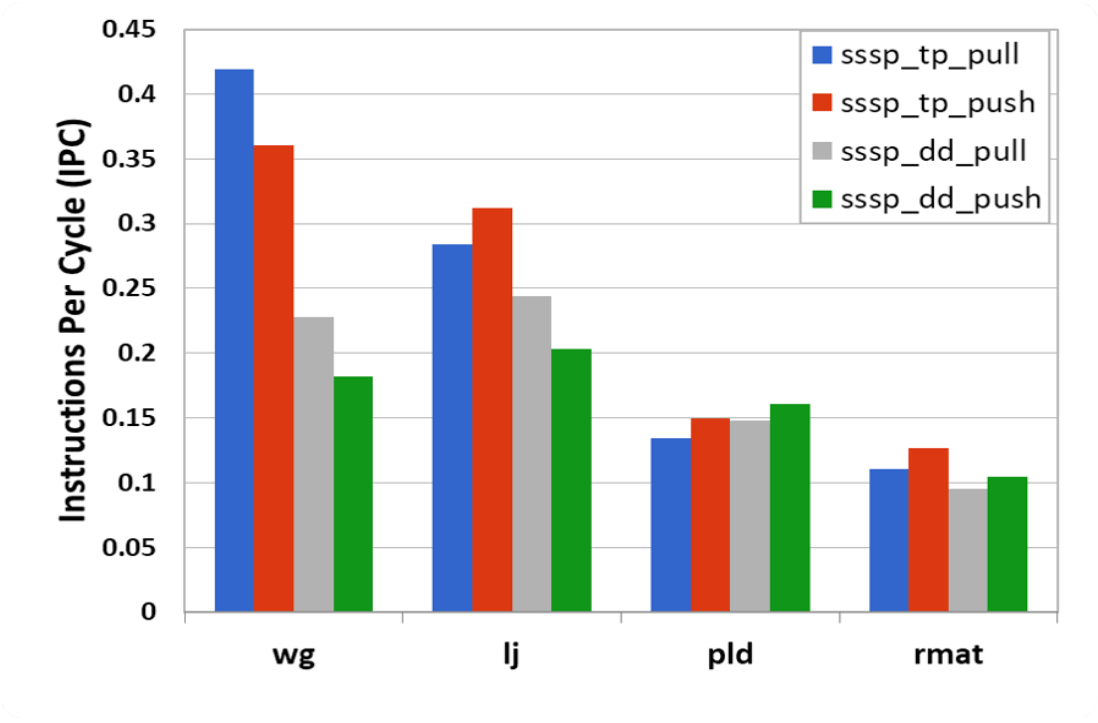

Figure 5.42: Instructions per cycle for SSSP with *wg*, *lj*, *pld*, and *rmat* graphs.

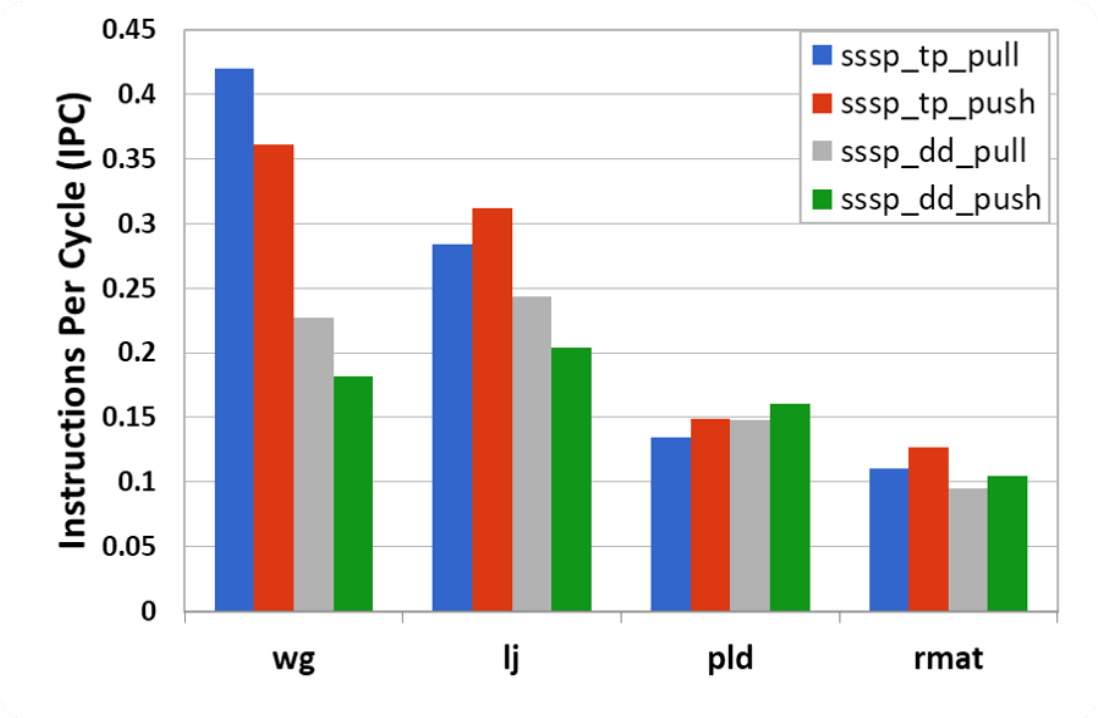

Figure 5.43: Instructions per cycle for BFS with *wg*, *lj*, *pld*, and *rmat* graphs.

Chapter 6

Discussion

We implemented eight different versions of PR algorithm by considering three design choices, namely, the order of computations, data access patterns, and work activation. Moreover, we implemented four different versions of SSSP and BFS by taking different combinations of data access patterns and work activations into account. We analyzed various combinations of different in-memory implementation styles of graph applications and explored the tradeoffs between performance, scalability, work efficiency, and computation cost of shared-memory systems.

Initially, we examine the performance and scalability of each application. First observation we made is that, the increase in the number of threads improves the performances of all algorithms as expected. For different data access patterns, work activation appears as a critical indicator which influences the performance and scalability of the algorithms. Moreover, we observed that data-driven push-based methods exhibit the highest performance amongst all the experiments. The least speedups are obtained from topology-driven pull-push-based methods. Nevertheless, work efficient pull-push based methods outperform their topology-driven versions by delivering substantial speedups. In addition, BFS implementations perform similar performance with the corresponding implementations of SSSP in terms of design decisions. For both classes, data-driven methods lead to a reduction in runtime in comparison with topology-driven methods, while fully

push-based alternatives outperform pull-based alternatives.

Secondly, we also noticed that if the activation of nodes is induced according to the structure of a graph, then the algorithm gives lower performance but high scalability. In contrast, if nodes are activated by utilizing a frontier, the algorithm performs well but it shows relatively low scalability in return. Next, when we investigated the decision of executing the applications with asynchronous execution model, the results demonstrated that if we execute a method in an asynchronous fashion, its performance improves. Although asynchronous executions require atomic operations, they are able to mitigate synchronization overheads by accelerating convergence rate of the algorithm.

Thirdly, we provided two different metrics in order to calculate work done such as total edges processed and total nodes activated. Our first observation was that data-driven push-based application is the most work efficient implementation among all applications. We noticed that data-driven push-based applications are performing much better due to the fact that they process the least number of edges and activate the least number of nodes. Moreover, data access patterns affect the number of edges processed when applications do not consider activation status of a node.

For determining micro-architectural bottlenecks, we reported cache behaviors of the aforementioned applications. In addition, we presented the rate of translation lookaside buffer (TLB) misses and the average number of instructions executed per cycle (IPC) for each application by using hardware performance counters.

Overall, optimizing graph applications for generally accepted metrics may not give the best performing implementation style. Let us consider scalability as an example. As noted before, pull-based implementations give the best scalability but it does not yield the best performance. On the other hand, data locality characteristics are mostly dependent on the size of datasets rather than details of the implementation. Moreover, for cache levels closer to main memory, the miss rates are quite high.

Furthermore, we observed a relation between data access patterns and work processing order and the performance. For all applications, choosing push based implementation combined with a frontier work activation model gives the best performance. In order to inspect the reason behind this, we have measured the amount of work performed by these implementations for which we have used number of edges processed as a proxy. These experiments revealed the fact that push based implementations are processing the minimum number of edges. Note that, "number of edges processed" metric is able to capture both the number of arithmetic/logical operations and the number of memory accesses.

We draw the following conclusions from the experimental results.

- If we want to optimize graph applications, we need to reduce memory accesses.
- Maximizing IPC values does not improve performance. Best IPC value gives the worst results.
- We did not observe a direct relation between data access pattern and L2/L3 miss rate for different algorithms, and L2/L3 miss rates are high for all design choices. Hence we conclude that these two metrics are not significant enough to identify the underlying cause of performance differences between algorithms. However, we observe that the graph size that the application is applied is directly proportional to L3 miss rates.
- Moreover, L1 and TLB miss rates are less than L2 and L3 miss rates. For the same dataset, dataset size, and layout but different access styles, L1 and TLB miss rates are different. In addition, patterns observed for L1 miss rates for each data access modes are also valid for TLB miss rates.

In light of our experiments, we propose that the work efficiency should be a first order metric for comparing graph applications. Furthermore, we propose a systematic approach to analyze this metric. As shown in Figure 3.1, there are three dimensions that cover the design space of graph applications. If we

consider the immediate gains and our practical experience of implementing these algorithms, we can suggest the following 4 step approach:

1. Implement baseline topology driven, pull based implementation
2. Introduce asynchronous execution model in order to leverage fast information propagation in the graph
3. Analyze different data access patterns: pull vs. push based implementations
4. Introduce a worklist implementation to enable data driven execution model

Note that, we have separated design choices into disjoint decisions. These steps enable programmers to stop when they achieve acceptable performance. According to our experiments with PR, SSSP, and BFS, one can achieve near optimal results when these four steps are followed.

Chapter 7

Conclusion

This thesis presents various combinations of different in-memory implementation styles of graph algorithms. We mainly focused on parallel iterative graph algorithms because many data analytics applications embody these properties. We observe that different implementations show different performance even on the same graph. For this purpose, in order to reveal these performance differences, large-scale graph algorithm design process is decomposed into three major components, and respectively analyzed by exploring the tradeoffs between runtime, scalability, work efficiency, and computation cost of shared-memory systems.

We analyzed behaviors of pull-based and push-based data access patterns. Our findings report that push-based implementations perform better for large datasets. Although their scalability is relatively less than their pull-based counterparts, these disadvantages can be ameliorated by using effective implementations of worklist structures and appropriate scheduling. Moreover, L2 and IPC metrics did not show significant changes for different implementations. Thus, other metrics should be defined in order to compare the performance of algorithms by considering different design choices.

In conclusion, we aims to provide a clear understanding of design metrics. These pointers can lead developers in designing high-performance large-scale

graph applications.

Bibliography

[1] G. A. Pavlopoulos, M. Secrier, C. N. Moschopoulos, T. G. Soldatos, S. Kossida, J. Aerts, R. Schneider, and P. G. Bagos, "Using graph theory to analyze biological networks," pp. 1–27, 2011.

[2] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web.," Technical Report 1999-66, Stanford InfoLab, November 1999.

[3] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, pp. 30–37, Aug 2009.

[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pp. 135–146, ACM, 2010.

[5] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "GraphLab: A new framework for parallel machine learning," *CoRR*, vol. abs/1006.4990, 2010.

[6] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning in the cloud," *CoRR*, vol. abs/1204.6078, 2012.

[7] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," *SIGPLAN Not.*, vol. 48, pp. 135–146, Feb. 2013.

[8] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pp. 211–222, ACM, 2007.

[9] A. Kusum, K. Vora, R. Gupta, and I. Neamtiu, "Efficient processing of large graphs via input reduction," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pp. 245–257, ACM, 2016.

[10] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015.

[11] L. Dhulipala, G. Blelloch, and J. Shun, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '17, pp. 293–304, ACM, 2017.

[12] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing Breadth-First Search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pp. 12:1–12:10, IEEE Computer Society Press, 2012.

[13] A. Lenharth, D. Nguyen, and K. Pingali, "Parallel graph analytics," *Communications of the ACM*, vol. 59, pp. 78–87, Apr. 2016.

[14] S. Beamer, *Understanding and Improving Graph Algorithm Performance*. PhD thesis, EECS Department, University of California, Berkeley, Sep 2016.

[15] G. Wang, W. Xie, A. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy," *CIDR*, 2013.

[16] D. Ediger, R. McColl, E. J. Riedy, and D. A. Bader, "STINGER: high performance data structure for streaming graphs," in *IEEE Conference on High Performance Extreme Computing, HPEC 2012, Waltham, MA, USA, September 10-12, 2012*, pp. 1–5, 2012.

[17] A. Kyrola, G. Blelloch, and C. 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, pp. 31–46, USENIX Association, 2012.

[18] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "CRONO: a benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *2015 IEEE International Symposium on Workload Characterization*, pp. 44–55, Oct 2015.

[19] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From 'think like a vertex' to 'think like a graph'," *Proceedings of the VLDB Endowment*, vol. 7, pp. 193–204, Nov. 2013.

[20] S. Salihoglu and J. Widom, "GPS: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pp. 22:1–22:12, ACM, 2013.

[21] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu, "TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pp. 77–85, ACM, 2013.

[22] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pp. 472–488, ACM, 2013.

[23] D. Zheng, D. Mhembere, R. C. Burns, and A. S. Szalay, "FlashGraph: Processing billion-node graphs on an array of commodity SSDs," *CoRR*, vol. abs/1408.0500, 2014.

[24] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proceedings of the USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pp. 375–386, USENIX Association, 2015.

[25] J. Zhong and B. He, "Medusa: Simplified graph processing on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, pp. 1543–1552, June 2014.

[26] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," *CoRR*, vol. abs/1501.05387, 2015.

[27] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "CuSha: Vertex-centric graph processing on GPUs," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pp. 239–252, ACM, 2014.

[28] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pp. 166–177, IEEE Press, 2016.

[29] T. Oguntebi and K. Olukotun, "GraphOps: A dataflow library for graph analytics acceleration," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pp. 111–117, ACM, 2016.

[30] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pp. 105–117, ACM, 2015.

[31] A. Buluç and J. R. Gilbert, "The combinatorial BLAS: design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[32] J. Seo, S. Guo, and M. S. Lam, "SociaLite: An efficient graph query language based on datalog," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, pp. 1824–1837, July 2015.

[33] D. P. Bertsekas, F. Guerriero, and R. Musmanno, "Parallel asynchronous label-correcting methods for shortest paths," *Journal of Optimization Theory and Applications*, vol. 88, pp. 297–320, Feb 1996.

[34] N. Sundaram, N. R. Satish, M. M. A. Patwary, S. Dulloor, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High performance graph analytics made productive," *CoRR*, vol. abs/1503.07241, 2015.

[35] A. Arasu, J. Novak, A. Tomkins, and J. Tomlin, "PageRank computation and the structure of the Web: Experiments and algorithms," in *Proceedings of the Eleventh International World Wide*, 2002.

[36] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, no. 1, pp. 269–271, 1959.

[37] U. Meyer and P. Sanders, "Δ-stepping : A parallel single source shortest path algorithm," in *Algorithms — ESA' 98* (G. Bilardi, G. F. Italiano, A. Pietracaprina, and G. Pucci, eds.), pp. 393–404, Springer Berlin Heidelberg, 1998.

[38] M. Kranjvcevi'c, D. Palossi, and S. Pintarelli, "Parallel delta-stepping algorithm for shared memory architectures," *CoRR*, vol. abs/1604.02113, 2016.

[39] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics XVI*, vol. XVI, no. 1, 1958.

[40] R. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pp. 1–11, IEEE Computer Society, 2010.

[41] R. W. Vuduc, *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, 2003. AAI3121741.

[42] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: a portable interface to hardware performance counters," in *Proceedings of the Department of Defense HPCMP Users Group Conference*, pp. 7–10, 1999.

[43] D. A. Bader, J. Berry, S. Kahan, R. Murphy, E. J. Riedy, J. Willcock, A. Korzh, and M. Zalewski, "Graph500." https://graph500.org/.

[44] R. Meusel, O. Lehmberg, C. Bizer, , and S. Vigna, "Web data commons." http://webdatacommons.org/hyperlinkgraph/.

[45] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." http://snap.stanford.edu/data, June 2014.

[46] J. J. Whang, A. Lenharth, I. S. Dhillon, and K. Pingali, *Scalable Data-Driven PageRank: Algorithms, System Issues, and Lessons Learned*, pp. 438–450. Springer Berlin Heidelberg, 2015.

[47] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Graph analytics accelerators for cognitive systems," *IEEE Micro*, vol. 37, pp. 42–51, Jan 2017.