STORAGE AND PROCESSING SYSTEMS FOR POWER-LAW GRAPHS

BY

IMRANUL HOQUE

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Associate Professor Indranil Gupta, Chair
Associate Professor ChengXiang Zhai
Professor Marc Snir
Dr. Malgorzata Steinder, IBM T. J. Watson Research Center

ABSTRACT

Large graphs abound around us – online social networks, Web graphs, the Internet, citation networks, protein interaction networks, telephone call graphs, peer-to-peer overlay networks, electric power grid networks, etc. Many real-life graphs are power-law graphs. A fundamental challenge in today's Big Data world is storage and processing of these large-scale power-law graphs.

In this thesis, we show that graph processing can be made faster and graph storage can be made more efficient by using techniques that leverage the structure of the underlying power-law graphs. To this end, we present two systems. First, we present LFGraph, which is a fast, distributed, in-memory graph analytics platform. LFGraph leverages the structure and characteristics of power-law graphs in order to reduce communication overhead, and to balance communication and computation load. This makes analytics faster on power-law graphs. Next, we present Bondhu, which is a disk layout manager for graph databases. Bondhu exploits the fact that most real-life power-law graphs are also small-world and these exhibit strong community structure. Bondhu utilizes this community structure in order to make layout decisions. This improves the query response time of graph databases. Our systems are evaluated on real clusters using real-world graphs.

To my family, for their love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

Graphs are an efficient way of encoding relationships among people, entities, and ideas. Some graphs occur in nature, e.g., protein interaction networks [1], metabolic networks [2], neural networks [3], food webs [4], etc. Additionally, some graphs evolve in man-made systems, e.g., online social networks [5, 6, 7], financial networks [8], function call graphs [9], Web graphs [10, 11], communication networks [12], transportation networks [13], etc. The sizes of these graphs range from a few thousand vertices and edges (e.g., a financial network) to billions of vertices and hundreds of billions of edges (e.g., the Facebook graph).

The degree distribution of many real-life graphs follows power-law, i.e., these graphs are scale-free. In these graphs, a vertex has degree d with probability proportional to $d^{-\alpha}$, $2 < \alpha < 3$. Therefore, a power-law graph consists of a few vertices of large degree, while a large number of vertices have relatively small degree. All of the above mentioned graphs are power-law.

Due to their unique degree distribution, power-law graphs exhibit some interesting characteristics. First, the high degree vertices (also known as hubs) are responsible for most of the edges in the graph. For example, in the Twitter graph the top 1% of the vertices are adjacent to 58% of the total edges [14]. Second, most real-life power-law graphs are also small-world. Small-world graphs exhibit a high clustering coefficient and show community structure. Therefore, in these graphs, the low degree vertices belong to clusters of densely connected sub-graphs. These clusters are interconnected

through the high degree vertices. So, most vertices can be reached from other vertices using only a few number of hops [15].

Today, we face two challenges associated with large-scale power-law graphs. First, there is an increasing demand for systems that can efficiently analyze these graphs. These analytics should be performed in an efficient manner, i.e., we desire fast result while using a small amount of resources. We present some examples of graph analytics here. Search engines such as Google [16], Bing [17], and Yahoo! [18] measure importance (i.e., ranking) of webpages by running PageRank computations on the Web graph. Online map services such as Google Maps [19], Bing Maps [20], and MapQuest [21] run shortest path computations on road networks in order to find fast routes. Matchmaking websites such as Chemistry.com, Match.com, and eHarmony run bi-partite matching computations on the social graph in order to find matches among users [22, 23, 24].

A second challenge is efficient storage of these graphs. It is imperative to make the graph storage systems more efficient, because efficiency in storage translates to high throughput and low latency in accessing the graph. Currently, graph databases are becoming popular as graph storage media. This is mainly due to three reasons: (i) graph databases lead to easier and intuitive representation and fast traversal of graphs, (ii) they make certain operations more efficient, e.g., finding degrees of separation, finding x hop neighbors, etc., and (iii) unlike relational databases they scale better to large graphs, e.g., they do not require expensive operations like joins. For instance, Twitter uses FlockDB to store social graphs with more than 13 billion edges (April 2010) [25]. We believe that efficient graph storage will be more criticial in coming years with the unprecedented growth of online social networking (OSN) sites and with contents from other online applications (e.g., shop-

ping, travel, review, media streaming, etc.) increasingly being integrated with OSNs,

1.1 Thesis Contributions

In this thesis, we show that techniques which leverage the structure of the power-law graph make graph computation faster and graph storage more efficient. We present the design and implementation of two systems. First, we present LFGraph[1], which is a distributed graph analytics platform. LFGraph makes graph computations faster while using a small amount of resources. LFGraph reduces the communication overhead of graph computations by leveraging the structure of power-law graphs into account. It also makes computations load balanced by leveraging the degree distribution characteristics of real power-law graphs. Second, we present Bondhu[2], which is a disk layout manager for graph databases. Bondhu's techniques lower the latency and increase the throughput for queries on graph databases. Bondhu makes disk placement decisions by leveraging the community structure of the underlying power-law OSN graph. This places frequently accessed data close by on disk and improves disk access performance.

Concretely, we make the following contributions in this thesis:

- Our first system, LFGraph, is the first distributed graph analytics framework to offer low and balanced communication and computation, low pre-processing overhead, low memory footprint, and scalability. LFGraph is primarily intended for directed graphs, although it can be adapted for undirected graphs. LFGraph uses a publish-

[1]Laissez-Faire Graph.
[2]Bangla word for friend.

subscribe based communication mechanism in order to reduce communication overhead and thus overall runtime for graph computations. LFGraph also shows that a random hash-based placement is sufficient to achieve communication and computation load balance for real power-law graphs. This reduces memory footprint of servers, lowers the number of servers needed to process a graph, and reduces total cost.

- In our second system, Bondhu, we present a novel framework for disk layout algorithms. Our techniques are based on community detection in graphs. Even though Bondhu is targeted at OSN graphs, it can be applied to any power-law small-world graph. First, we detect the communities within a social graph. Then, we produce the layout by running a greedy heuristic within and across the communities. To the best of our knowledge, Bondhu is the first system that leverages the social networking graph for efficient data layout in disks. We implement our solution into Neo4j, which is a widely-used open source graph database. While taking the community structure into account yields clear benefits, our results also indicate diminishing benefits from models that consider complexity beyond the graph structure itself.

- We evaluated our LFGraph system both with a Twitter graph (41 M vertices, 1.6 B edges) and a synthetic graph (1 B vertices and 127 B edges) in a 32 node Emulab cluster. Our results showed that LFGraph is upto 560 times faster than PowerGraph [26], which is the best existing graph analytics framework today. We also showed that LFGraph performs better that industrial system (e.g., Pregel [27]) using only a fraction of resources.

- We evaluated our Bondhu system with a real Facebook graph. We

showed that the Bondhu system is able to improve response time by as much as 48% compared to the default layout policy implemented by the file system.

1.2 Related Work

Graphs are characterized by two key aspects: vertex degree distribution and structural complexity. Vertex degree can follow uniform (e.g., regular graphs), exponential (e.g., random graphs), or power-law (e.g., scale-free graphs) distribution. Structural complexity generally refers to two important metrics: i) clustering co-efficient and ii) shortest-path length. For example, extended ring graphs have high clustering coefficient and high shortest path length. On the other hand, random graphs have low clustering coefficient and small shortest-path length. Small-world graphs have the best of these two types – a high clustering coefficient and a small shortest-path length.

Most of the biological, social, and man-made graphs are both power-law and small-world [28, 29]. Therefore, two major trends of works are existent in this area. First, there are works which have focused on how these graphs are dynamically generated, e.g., Watts-Strogatz model [30], preferential attachment model [31], etc. Second, there are systems which utilize the power-law and small-world nature of the graphs for improved performance. Examples include routing in small-world networks [32], searching in power-law graphs [33], spreading of information or epidemics [34], etc. This thesis falls in the second category.

Though most of the power-law graphs are small-world, there exist power-law graphs which are not small world. Instead of the preferential attachment model, these graphs follow the assortativity model, i.e., vertices tend to con-

nect with other vertices with similar degree [35]. The graph of global Avian Influenza outbreaks belongs to this category. Likewise there are graphs which are small-world, but not power-law. Examples include the acquaintance network of Mormons, the neuronal network of the work C. elegans, etc. In these graphs preferential attachment is limited by the age and the limited capacity of vertices [36].

1.3 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 discusses the design of LFGraph, a fast, scalable, distributed, in-memory graph analytics engine. The design of Bondhu, a social network-aware disk manager for the Neo4j graph database, is presented in Chapter 3. We present experimental evaluation of the LFGraph system in Chapter 4 and experimental evaluation of the Bondhu system in Chapter 5. We conclude by presenting our future directions in Chapter 6.

2 LFGRAPH: A DISTRIBUTED GRAPH ANALYTICS SYSTEM

In this chapter, we discuss LFGraph, a fast, scalable, distributed, in-memory graph analytics engine. LFGraph employs several power-law-aware optimizations in order to perform fast graph analytics. For example, LFGraph's publish-subscribe mechanism leverages the structure of the power-law graphs to reduce communication overhead. In addition, LFGraph utilizes the characteristics of real-world power-law graphs in order to offer load balanced computation and communication. LFGraph's techniques also lower resource utilization. Thus, LFGraph offers improved scalability compared to existing analytics frameworks.

This chapter presents the design of LFGraph and analytical results comparing it against existing systems. Later in Chapter 4 we discuss a cluster deployment of our implementation comparing it to the best system, PowerGraph. Our experiments used both synthetic graphs with a billion vertices, as well as several real graphs: Twitter, a Web graph, and an Amazon recommendation graph.

The rest of the chapter is organized as follows. Section 2.1 motivates the design decisions of the LFGraph system. Section 2.2 discusses the computation model and API adopted by LFGraph, compares the LFGraph abstraction with existing models, and presents three sample programs written using LFGraph's API. We present the design of the LFGraph system in Section 2.3. We analytically compare the communication overhead of LFGraph with those of existing systems in Section 2.4. Section 2.5 shows that LFGraph is able

to offer communication and computation load balance by utilizing the characteristics of real-life power-law graphs. Finally, we present related work in Section 2.7.

2.1 Motivation

Distributed graph processing frameworks are being increasingly used to perform analytics on the large graphs that surround us today. A large number of these graphs are directed power-law graphs, such as follower graphs in online social networks, the Web graph, recommendation graphs, financial networks, and others. These graphs may contain millions to billions of vertices, and hundreds of millions to billions of edges.

Systems like Pregel [27], GraphLab [37], GraphChi [38], and PowerGraph [26] are used to compute metrics such as PageRank and shortest path, and to perform operations such as clustering and matching. These frameworks are vertex-centric and the processing is iterative. In each iteration (called a *superstep* in some systems) each vertex executes the same code and then communicates with its graph neighbors. Thus, an iteration consists of a mix of computation and communication.

We believe that a distributed graph analytics engine running in a cluster must pay heed to five essential aspects:

1. *Computation:* The computation overhead must be low and load-balanced across servers. This determines per-iteration time and thus overall job completion time. It is affected by the number and distribution of vertices and edges across servers.

2. *Communication:* Communication overhead must be low and load-balanced across servers. This also determines per-iteration time and thus

overall job completion time. It is affected by the quantity and distribution of data exchanged among vertices across servers.

3. *Pre-Processing*: Prior to the first iteration, the graph needs to be partitioned across servers. This partitioning time must be low since it represents upfront cost and is included in job completion time.

4. *Memory*: The memory footprint per server must be low. This ensures that fewer servers can be used for processing large graphs, e.g., when resources are limited.

5. *Scalability*: Smaller clusters must be able to load and process large graphs. As the cluster size is grown, communication and computation must become cheaper, and the entire job must run faster.

Unfortunately, each of today's graph processing frameworks falls short in at least one of the above categories. We will elaborate later in Section 2.2.3, and also experimentally compare against existing systems. For now, Table 2.1 summarizes a qualitative comparison, and we briefly discuss. GraphChi [38] is a disk-based single-server framework – so, it is slower than distributed frameworks. Pregel [27] was the first vertex-centric distributed graph processing framework. It suffers from both high memory footprint and high communication overhead. GraphLab [37] and PowerGraph [26] have lower communication overhead compared to Pregel, and PowerGraph also balances computation. They are both faster than Pregel. However, these latter systems store in-links and out-links for each vertex, thus increasing the memory footprint. They are thus unable to process large graphs on small clusters.

The fastest of these systems, PowerGraph, uses intelligent partitioning of vertices across servers. While this pre-processing reduces per iteration

Goal	Pregel	GraphLab	PowerGraph	LFGraph
Computation	2 passes, combiners	2 passes	2 passes	1 pass
Communication	$\propto$ #edge-cuts	$\propto$ #vertex ghosts	$\propto$ #vertex mirrors	$\propto$ #external in-neighbors
Pre-processing	Cheap (Hash)	Cheap (Hash)	Expensive (Intelligent)	Cheap (Hash)
Memory	High (store out-edges + buffered messages)	High (store in- and out-edges)	High (store in- and out-edges)	Low (store in-edges)
Scalability	Good but needs min #servers	Good but needs min #servers	Good but needs min #servers	Good and runs with small #servers

Table 2.1: LFGraph vs. existing systems: a qualitative comparison

runtime, it is an expensive step. For instance, we found that when running PageRank on PowerGraph with 8 servers and 30 iterations (a value that Pregel uses [27]), the intelligent partitioning step constituted 80% of the total job runtime. This upfront cost might make sense if it is amortized over multiple analytics jobs on the same graph. However, we show that cheaper partitioning approaches do not preclude faster iterations.

This chapter presents LFGraph, the first system to satisfy the five requirements outlined earlier. LFGraph is a fast, scalable, distributed, in-memory graph analytics framework. LFGraph's key design choices are motivated by the fact that most of the graphs processed by today's graph analytics frameworks are power-law in nature. Therefore, techniques which leverage the structure and characteristics of the underlying power-law graphs make graph analytics faster. The unique design choices in our system are:

- **Cheap Partitioning**: We rely merely on hash-based partitioning of vertices across servers. Substantial variability in the degree distribution of the high degree vertices of the power-law graphs helps LFGraph

achieve balanced communication and computation. This approach lowers pre-processing overhead and system complexity.

- **Publish-Subscribe Mechanism**: Most graph computations involve information flow along its directed edges. In power-law graphs a small number of vertices are adjacent to a large number of vertices. Therefore, a small number of vertices is responsible for majority of the communication overhead. So, in case of a vertex which has multiple friends at a remote server, communication overhead can be significantly reduced if only one copy of the message is sent to the remote server and the friends share that message internally. Using this observation, we create a publish-subscribe based communication mechanism which creates a fetch-once communication pattern. This leads to significant savings, e.g., compared to PowerGraph [26], LFGraph reduces network traffic by 4x.

- **Decoupling Computation from Communication:** This leads to modular code. It also allows us to optimize communication and computation independent of each other.

- **Single-pass Computation**: The per-iteration computation at each server is done in one pass, resulting in low computation overhead. Each of Pregel, PowerGraph, and GraphLab uses multiple passes. Pregel incurs the additional overhead of message combiners. LFGraph is simpler and yet its individual iterations are faster than in existing systems.

- **No Locking:** LFGraph eliminates locking by decoupling reads and writes to a vertex's value.

- **In-neighbor Storage**: LFGraph maintains for each vertex only its

in-neighbors. Compared to existing systems which maintain both in- and out-neighbors, LFGraph lowers memory footprint and is thus able to run large graphs even on small clusters. LFGraph is extendible to undirected graphs by treating each edge as two directed edges.

2.2 Computation Model

This section presents the assumptions LFGraph makes, the LFGraph abstraction, and a qualitative comparison with existing systems. Then we present LFGraph's API and sample graph processing applications using this API.

2.2.1 Assumptions

- LFGraph performs computations on the graph itself rather than performing data mining operations on graph properties such as user profile information.
- LFGraph framework is intended for value propagation algorithms. Values propagate along the direction of the edges. Algorithms that fall in this category include PageRank, Single Souce Shortest Path, Triangle Counting, Matching, Clustering, Graph Coloring, etc.
- LFGraph assumes that the number of high degree vertices is much larger than the number of servers. This is necessary to achieve load balance (see Section 2.4.2) and to reduce communication overhead.

2.2.2 LFGraph Abstraction

An LFGraph server stores each graph vertex as a tuple ⟨vertex ID, user-defined value⟩. The type of the user-defined value is programmer-specified,

e.g,. in PageRank it is a floating point, for single-source shortest path (SSSP) it is an integer, and for triangle counting it is a list. For each vertex a list of *incoming* edges is maintained. An edge is also associated with a user defined value that is typically static, e.g., the edge weight.

Abstraction 2.1 LFGraph

```
1: function LFGraph(Vertex v)
2:     val[v] ← f(val[u]), u ∈ in_neighbor(v)
3: end function
```

LFGraph uses the programming model shown in Abstraction 2.1. The programmer writes a vertex program $f()$. This program runs in iterations, akin to supersteps in existing systems [26, 37, 27]. Each vertex is assigned to one server. The start of each iteration is synchronized across servers. During an iteration, the vertex program for vertex v reads the values of its incoming neighbors, performs the computation specified by $f()$, and updates its own value. If v's value changes during an iteration, it is marked as active, otherwise it is marked as inactive. The framework transmits active values to the servers containing neighboring vertices. The computation terminates either at the first iteration when all vertices are inactive (e.g., in SSSP), or after a pre-specified number of iterations (e.g., in PageRank).

2.2.3 Qualitative Comparison

The abstractions employed by Pregel, GraphLab, and PowerGraph are depicted respectively in Abstraction 2.2, 2.3, and 2.4. To contrast with LFGraph we first discuss each of these systems and then summarize LFGraph. We use a running example below (Figure 2.1). Table 2.1 summarizes this discussion.

Pregel: Pregel assigns each vertex to one server. Per iteration, v's vertex

Abstraction 2.2 Pregel

1: **function** PREGEL(Vertex v)
2: $val[v] \leftarrow f(msg)$, $sender(msg_i) \in in\_neighbor(v)$
3: $send\_message(val[v], u)$, $u \in out\_neighbor(v)$
4: **end function**

Abstraction 2.3 GraphLab

1: **function** GRAPHLAB(Vertex v)
2: $val[v] \leftarrow f(val[u])$, $u \in in\_neighbor(v)$
3: **if** $updated(val[v])$ **then**
4: $activate(u)$, $u \in out\_neighbor(v)$
5: **end if**
6: **end function**

Abstraction 2.4 PowerGraph

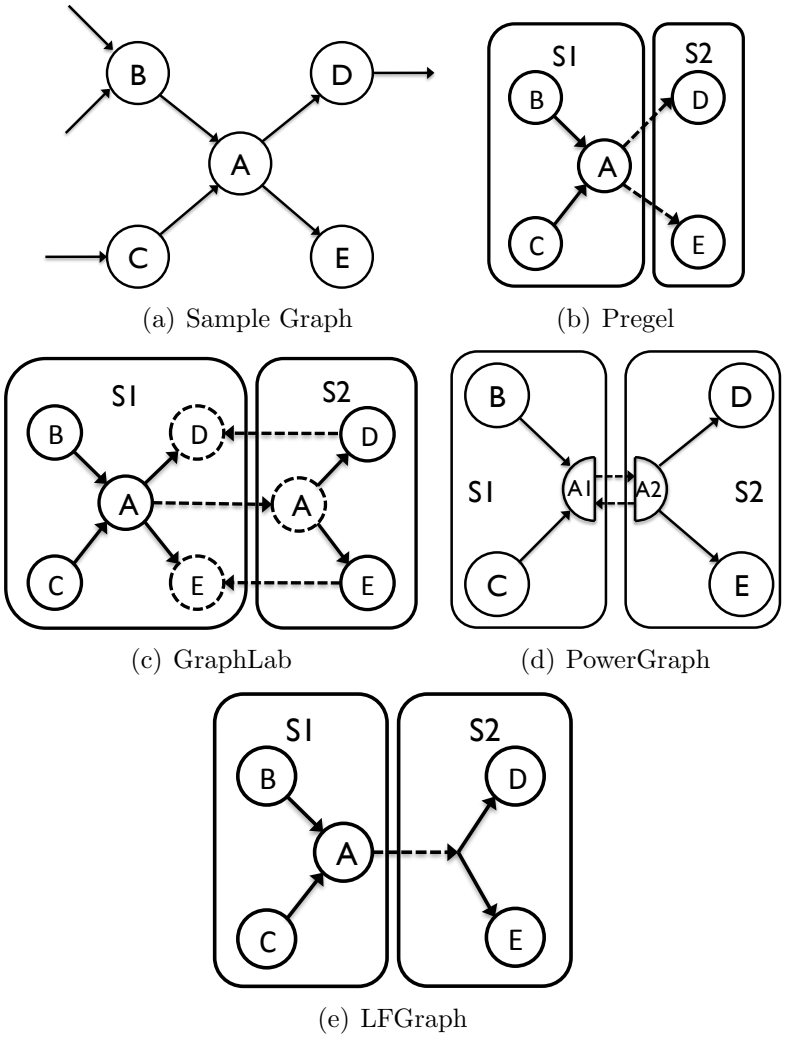
1: **function** POWERGRAPH(Vertex v_i)
2: $val[v_i] \leftarrow f(val[u])$, $u \in in\_neighbor(v_i)$
3: $val[v] \leftarrow sync(v_i)$, $v_i \in replica(v)$
4: **if** $updated(val[v])$ **then**
5: $activate(u)$, $u \in out\_neighbor(v_i)$
6: **end if**
7: **end function**

program uses its received neighbor values to update the vertex value, and then sends this new value back out to servers where v's neighbors are located.

Consider the sliver of the graph depicted in Figure 2.1(a). We focus on the vertex program for A only, and our example cluster contains two servers $S1$ and $S2$. Figure 2.1(b) shows that Pregel's communication overhead (dashed arrows) is proportional to the number of edges crossing server boundaries – A's value is sent twice from $S1$ to $S2$, once for each neighbor. Pregel does allow programmers to write combiners to optimize communication, but this increases computation complexity by requiring an additional pass over the outgoing messages. Besides, some analytics programs do not lend themselves easily to combiners.

GraphLab: GraphLab first assigns each vertex (say A) to one server ($S1$). Then for each of A's in- and out- neighbors not assigned to $S1$, it creates *ghost* vertices, shown as dashed circles in Figure 2.1(c). A is assigned to $S1$ but is ghosted at $S2$ since its out-neighbor D is there. This allows all edge communication to avoid the network, but at the end of the iteration all the ghosts of A need to be sent its new value from A's main server ($S1$). This means that GraphLab's communication overhead is proportional to the number of ghosts. However, the number of ghosts can be very large – it is bounded by min(cluster size, total number of in- and out-neighbors). Section 2.4 shows that this leads to high communication overhead when processing real graphs with high degree vertices.

If A's value at a server is updated during an iteration, GraphLab activates its outgoing neighbors (lines 3–5 in Abstraction 2.3). This requires GraphLab to store both in- and out- neighbor lists, increasing memory footprint. Further, per vertex, two passes are needed over its in- and out- neighbor lists. The first pass updates its value, and the second activates the out-neighbors.



(a) Sample Graph (b) Pregel

(c) GraphLab (d) PowerGraph

(e) LFGraph

Figure 2.1: Communication overhead

PowerGraph: In order to target power-law graphs, PowerGraph places each *edge* at one server. This means that vertex A may have its edges placed at different servers. Thus PowerGraph creates *mirrors* for A at $S1$ and $S2$, as shown in Figure 2.1(d). The mirrors avoid edge communication from crossing the network. However, the mirrors need to aggregate their values during the iteration. PowerGraph does this by designating one of the mirrors as a master. In the middle of the iteration (line 3 of Abstraction 2.4), all A's mirrors send their values to its master ($A1$), which then aggregates them and sends them back. Thus, communication overhead is proportional to

twice the number of vertex mirrors, which can be very large and is bounded by min(cluster size, total number of in- and out-neighbors). We show in Section 2.4 that PowerGraph incurs high communication overhead for real graphs.

LFGraph: As depicted in Figure 2.1(e), LFGraph assigns each vertex exactly to one server (A at $S1$). LFGraph makes a single pass over the in-neighbor list of A – this reduces computation. $S1$ stores only a publish list of servers where A's out-neighbors are placed (only $S2$ here), and uses this to forward A's updated value. This leads to the fetch-once behavior at $S2$, thus reducing communication significantly for vertices with a large number of out-neighbors (hubs in power-law graphs). The publish list is upper-bounded by min(cluster size, total number of out-neighbors), which is smaller than the number of ghosts or mirrors in GraphLab and PowerGraph respectively – thus LFGraph's memory footprint is smaller, communication overhead is lower, and it works even in small clusters. Section 2.3 elaborates further on the design, and we analyze it in Section 2.4.

LFGraph trades off computation for reduced storage – in an iteration, it needs to run through all the vertices to check if any of them is in fact active. In contrast, PowerGraph and GraphLab have activate/deactivate triggers which can enable/disable the execution of a neighboring vertex in the succeeding iteration.

2.2.4 LFGraph API

The programmer writes an LFGraph program which uses LFGraph's Vertex class. The exported methods of the Vertex class (simplified) are depicted in Table 2.2. We show how these methods can be used to write three graph

Function	Description
getInLinks()	returns a list of in-edges
getUpdatedInLinks()	returns a list of in-edges whose source vertices updated in the previous iteration
int getOutLinkCount()	returns the count of out-edges
getValue(int vertexID)	returns the value associated with vertexID
putValue(VertexValue value)	writes updated value
int getStep()	get iteration count

Table 2.2: LFGraph API: Vertex class methods

analytics program: PageRank [39], SSSP (single-source shortest path), and triangle counting [40].

PageRank Vertex Program

```
1:  if getStep() = 0 then
2:      putValue(1)
3:  else if getStep() < 30 then
4:      total ← 0
5:      for e ∈ getInLinks() do
6:          v ← e.getSource()
7:          total ← total + getValue(v)
8:      end for
9:      pagerank ← (0.15 + 0.85 × total)
10:     putValue(pagerank/getOutLinkCount())
11: end if
```

PageRank Each vertex sets its initial PageRank to 1 (line 1–2). In subsequent iterations each vertex obtains its in-neighbors' values via $getValue()$ (line 5–8), calculates its new PageRank (line 9), and updates its value using $putValue()$ (line 10). The LFGraph system is responsible for transferring the values to the appropriate servers.

SSSP In the first iteration, only the source vertex sets its value (distance) to 0 while all others set their value to ∞ (line 2–6). During subsequent iterations a vertex reads the value of its *updated* in-neighbors, calculates the minimum distance to the source through all of its in-neighbors (line 9–13),

SSSP Vertex Program

```
1: if getStep() = 0 then
2:     if vertexID = srcID then
3:         putValue(0)
4:     else
5:         putValue(∞)
6:     end if
7: else
8:     min_dist ← ∞
9:     for e ∈ getUpdatedInLinks() do
10:        v ← e.getSource()
11:        dist ← getValue(v) + e.getValue()
12:        min_dist ← min(min_dist, dist)
13:    end for
14:    if getValue(vertexID) > min_dist then
15:        putValue(min_dist)
16:    end if
17: end if
```

and updates its value if the minimum distance is lower than its current value (line 14–16). LFGraph only transfers a vertex's value if it was updated during the previous iteration.

Triangle Counting This works on an undirected graph, so $getInLinks()$ returns all neighbors of a vertex. In the first iteration, each vertex initializes its *value* to the list of its neighbors (line 1–2) . In the second iteration, a vertex calculates, for each of its neighbors, the number of their common neighbors (line 6–10). The final answer is obtained by dividing the count by 2, since triangles are double-counted (line 11).

2.3 System Design

The LFGraph system consists of three components:

1. *Front-end*: The front-end stores the vertex program and configuration information. The only coordination it performs is related to fault tol-

TriangleCount Vertex Program

```
1: if getStep() = 0 then
2:     putValue(getInLinks())
3: else
4:     count ← 0
5:     s1 ← getValue(vertexID)
6:     for e ∈ getInLinks() do
7:         v ← e.getSource()
8:         s2 ← getvalue(v)
9:         count ← count + set_intersect(s1, s2)
10:    end for
11:    count ← count/2
12: end if
```

erance (Section 2.6).

2. *JobServers*: A single JobServer runs at each server machine. A JobServer is responsible for loading and storing the part of the graph assigned to that server, and launching the vertex-program. The JobServer is composed of four modules, which subsequent sections detail: i) Graph Loader, ii) Storage Engine, iii) Computation Workers, and iv) Communication Workers. Each iteration at a JobServer consists of a *computation phase* run by several computation workers, followed by a decoupled *communication phase* performed by several communication workers.

3. *BarrierServer:* This performs distributed barrier synchronization among JobServers at three points: i) after loading the graph, ii) during each iteration in between the computation and communication phases, and iii) at the end of each iteration.

Figure 2.2 shows an example execution of an LFGraph iteration. Steps 1 – 3 comprise the pre-processing iteration (graph loading), and steps 4 – 6 are repeated for each iteration. We elaborate on each step:

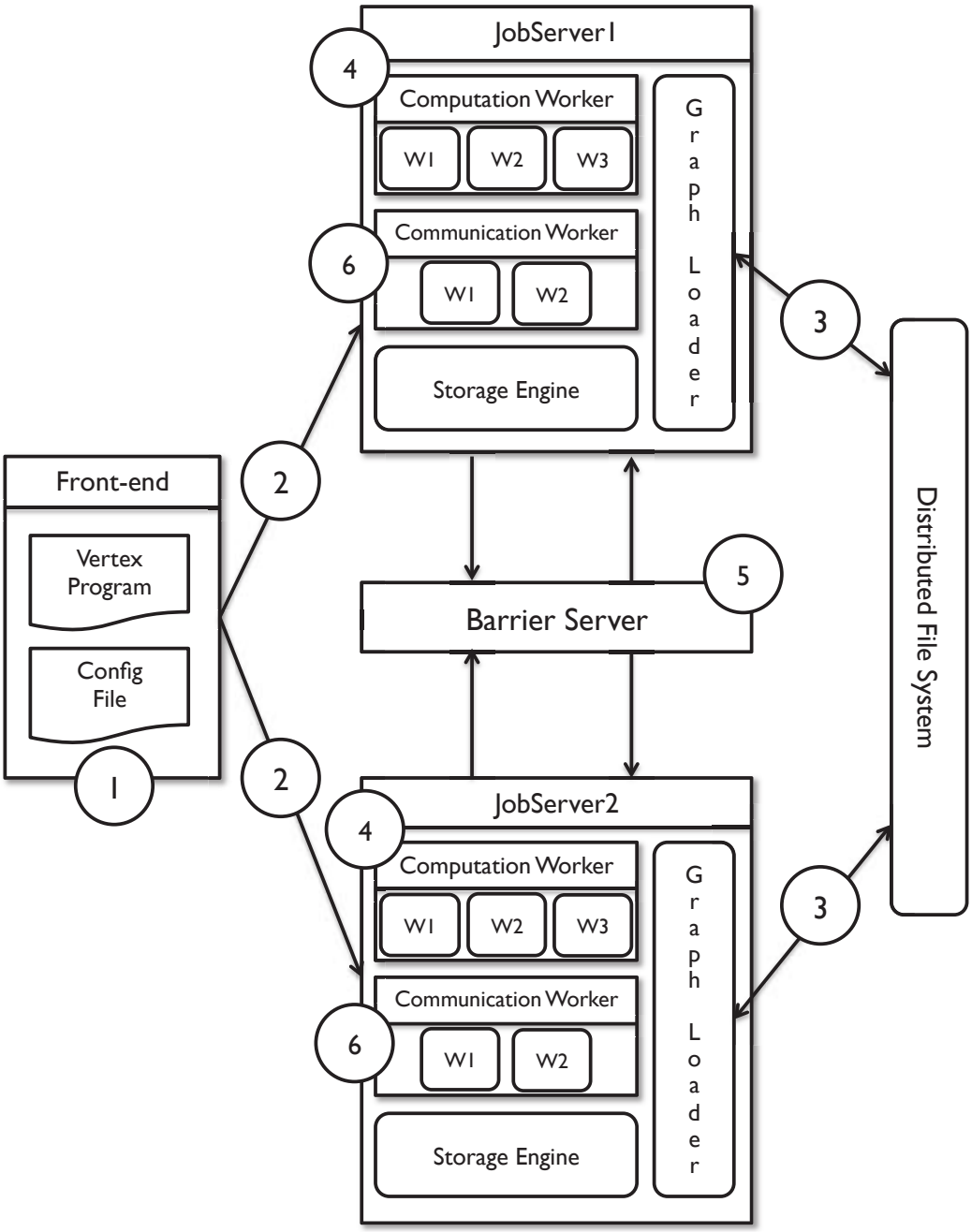


Figure 2.2: LFGraph system: the life of an iteration

1. The front-end stores the vertex program and a configuration file containing the following pieces of information: graph data location (e.g., an NFS path), number of JobServers, IP addresses of JobServers, IP address of BarrierServer, number of computation and communication

workers per JobServer, and (if applicable) maximum number of iterations desired.

2. The front-end sends the vertex program and configuration file to all JobServers.

3. The Graph Loaders collectively load the graph (Section 2.3.1) and split them across the Storage Engines (Section 2.3.2). The JobServer then signals *next* to the BarrierServer.

   Each iteration repeats the following steps 4 – 6.

4. When the BarrierServer signals back that the barrier is completed, a JobServer starts the next iteration. It does so by spawning multiple local computation worker threads (Section 2.3.3) to start the computation phase, and sending each worker a sub-shard of its vertices.

5. When all computation workers finish, the JobServer signals the BarrierServer. The signal is one of two types: *next* or *halt*. The latter is signaled only if the termination conditions are met, e.g., maximum number of iterations desired has been reached (e.g., in PageRank) or no local vertices have updated their value in the computation phase (e.g., in SSSP). The BarrierServer terminates the job only if all JobServers signal *halt*.

6. If any of the JobServers signaled *next*, the BarrierServer signals back when the barrier is reached. Then the JobServer starts the communication phase by spawning communication worker threads (Section 2.3.4). Communication workers are responsible for sending vertex values to remote JobServers. Then it signals *next* to the BarrierServer to start the next iteration.

We next detail the four modules inside the JobServer.

2.3.1 Graph Loader

Graph Loaders are collectively responsible for loading and partitioning vertex data from the distributed file system (e.g., NFS) and sending them to JobServers. LFGraph can accept a variety of input formats, e.g., edge list, out-neighbor adjacency list, in-neighbor adjacency list, etc. The data is sharded across Graph Loaders. A Graph Loader first uses the path provided by the front-end to load its assigned set of vertices. For each line it reads, it uses a consistent hash function on the vertex ID to calculate that vertex's JobServer, and transmits the line to that JobServer. For efficiency, the Graph Loader batches several vertices together before sending them over to a JobServer.

2.3.2 Storage Engine

This component of the JobServer stores the graph data, uses a publish-subscribe mechanism to enable efficient communication, and stores the vertex values. It maintains the following data structures.

Graph Storage: This stores the list of in-neighbors for each vertex assigned to this JobServer. $getInLinks()$ in Table 2.2 accesses this list.

Subscribe Lists: This maintains a short-lived list per remote JobServer. Each such list contains the vertices to be fetched from that specific JobServer. The list is built only once – at the pre-processing iteration while loading the graph. Consider our running example from Figure 2.1(e). $S2$'s subscribe list for $S1$ consists of $\{A\}$. The subscribe list is short-lived because it is garbage-collected after the preprocessing iteration, thus reducing memory

usage.

Publish Lists: A JobServer maintains a publish list for each remote JobServer, containing the vertices to be sent to that JobServer. Publish lists are intended to ensure that each external vertex data is fetched exactly once. In the pre-processing iteration, JobServers exchange subscribe lists and use them to create publish lists. In our example, JobServer $S2$ sends to $S1$ its subscribe list for JobServer $S1$. Then the publish list at $S1$ for $S2$ contains those vertices assigned to $S1$ which have out-neighbors assigned to $S2$, i.e., the set $\{A\}$.

Local Value Store: For each vertex assigned to this JobServer (call this a local vertex), this component stores the value for that vertex. We use a two-version system for each value – a *real version* from the previous iteration and a *shadow version* for the next iteration. Writes are always done to the shadow value and reads always occur from the real value. At the end of the iteration, the real value is set to the shadow value, and the latter is un-initialized. The shadow is needed because computation workers at a JobServer share the local value store. Thus a vertex D at JobServer $S2$ may update its local value, but later another local vertex E also at $S2$ needs to read D's value. Further, this two-version approach decouples reading and writing, thus eliminating the need for locking.

Each value in the local value store is also tagged with an $updated_{A,S1}$ flag, which is reset at the start of an iteration. Whenever the shadow value is written, this flag is set. The communication worker component (which we describe later) uses this flag.

Remote Value Store: This stores the values for each in-neighbor of a vertex assigned to this JobServer, e.g., at JobServer $S2$, the remote value store contains an entry for remote vertex A. There is only one version here since

it is only used for reading. This value is also tagged with a flag $updated_{A,S2}$ which is reset at the start of the communication phase – the flag is set only if $S2$ receives an updated value for A during the current communication phase, otherwise it is left unset. This information is used to skip vertices, whose values did not update, in the upcoming iteration. The $getUpdatedInLinks()$ function (in Table 2.2) of the vertex uses the update flags to return the list of neighbors whose values were updated.

We briefly discuss memory implications of the above five stores of the Storage Engine. Any graph processing framework will need to store the graph and the local value store. The subscribe list is garbage collected. Thus LFGraph's additional overheads are only the publish list and the remote value store. The latter of these dominates, but it stays small in size even for large graphs. For a log-normal graph with 1 billion vertices and 128 billion edges in a cluster of 12 machines running the SSSP benchmark, the per-JobServer remote value store was smaller than 3.5 GB.

2.3.3 Computation Worker

A computation worker is responsible for running the front-end-specified vertex program sequentially for the sub-shard of vertices assigned to it. Our implementation uses a thread pool for implementing the workers at a JobServer. The number of computation workers per JobServer is a user-specified parameter. For homogeneous clusters, we recommend setting this value to the number of cores at a server.

The computation worker accesses its JobServer's local value store and remote value store. Yet, no locking is required because of the sub-sharding and the two-versioned values. For each vertex this worker reads its in-neighbors'

data from either the remote or local value store (real versions only), calculates the new value and writes its updated value into the local value store (shadow version).

2.3.4 Communication Worker

A communication worker runs the decoupled communication phase. It does not rely on locking. The worker runs in four phases. First, each worker is assigned a sub-shard of remote value stores. It resets the update flags in this sub-shard. Second, the worker is assigned a sub-shard of remote JobServers. For each assigned remote JobServer, the worker looks up its publish list, and then sends the requisite vertex values. It uses shadow values from the local value store, skipping vertices whose update flags are false. Third, when a JobServer receives a value from a remote JobServer, it forwards this to the appropriate local worker, which in turn updates the remote value store and sets the update flags. These second and third phases are overlapped. Fourth and finally, the worker is assigned a sub-shard of the local vertices, for which it moves the shadow value to the real value.

We use a thread pool for the communication workers. Communication workers in differents machines use sockets for interprocess communication.

2.4 Communication Analysis

We first present mathematical analysis for the communication overhead of LFGraph and existing graph processing frameworks (Section 2.4.1). Then we use three real-world graphs (Twitter, a Web graph, and an Amazon recommendation graph) to measure the realistic impact of this analysis (Section 2.4.2) and compare these systems. Although we will later present experi-

mental results from our deployment (Section 4), the analysis in this section is the most appropriate way to compare the fundamental *techniques* employed by different systems. This analysis is thus independent of implementation choices (e.g., C++ vs. Java), optimizations, and myriad possible system configurations.

2.4.1 Mathematical Analysis

Define the communication overhead of a given vertex v as the expected number of values of v sent over the network in a given iteration. We assume all vertex values have changed, thus the metric is an upper bound on the actual average per-vertex communication. Then, define the communication overhead of an algorithm as the average of the communication overheads across all vertices. We assume the directed graph has $|V|$ vertices, and the cluster contains N servers ($V >> N$). We denote the out-neighbor set of a vertex v as $D_{out}[v]$ and its in-neighbor set as $D_{in}[v]$. We also assume that values are propagated in one direction and values are of fixed sizes.

Pregel:

In a default (combiner-less) Pregel setting, each vertex is assigned to one server. Thus values are sent along all edges. An edge contributes to the communication overhead if its adjacent vertices are on different servers. An out-neighbor of v is on a different server than v with probability $\left(1 - \frac{1}{N}\right)$. The communication overhead of v is thus:

$$C_P(v) = |D_{out}[v]| \times \left(1 - \frac{1}{N}\right) \tag{2.1}$$

Therefore, Pregel's communication overhead is:

$$C_P = \frac{\sum_{v \in V} \left(|D_{out}[v]| \times \left(1 - \frac{1}{N}\right) \right)}{|V|} \tag{2.2}$$

GraphLab:

In GraphLab, each vertex is assigned to a server. However, the vertex has multiple ghosts, one at each remote server. A ghost is created at remote server S if at least one of v's in- or out-neighbors is assigned to S. The main server where v is assigned then collects all the updated values from its ghosts. v has no neighbors at a given remote server with probability $\left(1 - \frac{1}{N}\right)^{|D_{out}[v] \cup D_{in}[v]|}$. Thus the probability that v has at least one of its neighbors at that remote server is: $\left(1 - (1 - \frac{1}{N})^{|D_{out}[v] \cup D_{in}[v]|}\right)$. Hence the communication overhead of v is:

$$C_{GL}(v) = (N-1) \times \left(1 - \left(1 - \frac{1}{N}\right)^{|D_{out}[v] \cup D_{in}[v]|}\right) \tag{2.3}$$

The communication overhead of GraphLab is:

$$C_{GL} = \frac{\sum_{v \in V} \left((N-1) \times \left(1 - \left(1 - \frac{1}{N}\right)^{|D_{out}[v] \cup D_{in}[v]|}\right)\right)}{|V|} \tag{2.4}$$

PowerGraph:

In PowerGraph, each vertex is replicated (mirrored) at several servers – let r_v denote the number of replicas of vertex v. One of the replicas is designated as the master. The master receives updated values from the other $(r_v - 1)$

Graph	Description	$\|V\|$	$\|E\|$
Twitter	Twitter follower network [14]	41.65M	1.47B
UK-2007	UK Web graph [41]	105.9M	3.74B
Amazon	Similarity among books in Amazon store [42, 43]	0.74M	5.16M

Table 2.3: Real-world graphs

replicas, calculates the combined value, and sends it back out to the replicas. Thus, the communication overhead of v is $2 \times (r_v - 1)$. Plugging in the value of r_v from [26], the communication overhead of PowerGraph is:

$$\begin{aligned} C_{PG} =& \frac{2 \cdot \sum_{v \in V} (r_v - 1)}{|V|} \\ =& \frac{2 \cdot \sum_{v \in V} \left(N \times \left(1 - \left(1 - \frac{1}{N} \right)^{|D_{out}[v] \cup D_{in}[v]|} \right) - 1 \right)}{|V|} \end{aligned} \tag{2.5}$$

LFGraph:

In LFGraph, a vertex v is assigned to one server. Its value is fetched *by* a remote server S if at least one of v's *out*-neighbors is assigned to S. v has at least one out-neighbor at S with probability $\left(1 - (1 - \frac{1}{N})^{|D_{out}[v]|}\right)$. The communication overhead of v is:

$$C_{XG}(v) = (N-1) \times \left(1 - \left(1 - \frac{1}{N} \right)^{|D_{out}[v]|} \right) \tag{2.6}$$

Therefore, LFGraph's communication overhead is:

$$C_{XG} = \frac{\sum_{v \in V} \left((N-1) \times \left(1 - \left(1 - \frac{1}{N} \right)^{|D_{out}[v]|} \right) \right)}{|V|} \tag{2.7}$$

Discussion

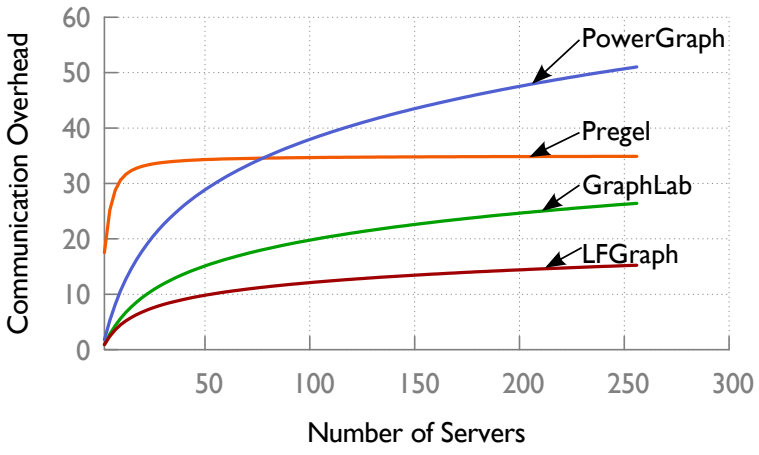
Our analysis yields the following observations:

- The overheads of Pregel and LFGraph depend on $|D_{out}[v]|$ only, while those of GraphLab and PowerGraph depend on $|D_{out}[v] \cup D_{in}[v]|$.
- For an undirected graph $|D_{out}[v] \cup D_{in}[v]| = |D_{out}[v]| = |D_{in}[v]|$, so communication overhead of LFGraph and GraphLab are similar for such graphs.
- PowerGraph is a factor of 2 worse in communication overhead compared to GraphLab.
- LFGraph has its lowest relative communication overhead when $|D_{out}[v] \cup D_{in}[v]| \gg |D_{out}[v]|$, i.e., when out-neighbor and in-neighbor sets are more disjoint from each other, and the in-degree is larger than the out-degree.

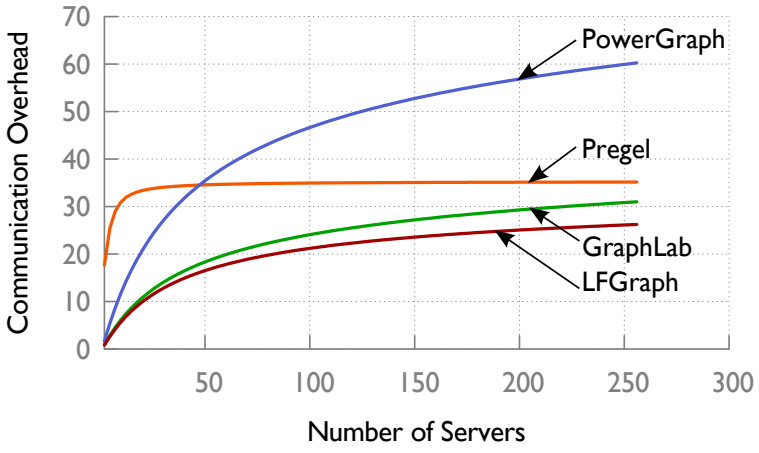
2.4.2 Real-World Graphs

We now study the impact of the previous section's analysis on several real-world directed graphs: 1) Twitter, 2) a graph of websites in UK from 2007, and 3) a recommendation graph from Amazon's online book store. The characteristics of these are summarized in Table 2.3. The traces contain a list of vertices and out-neighbor adjacency lists per vertex.

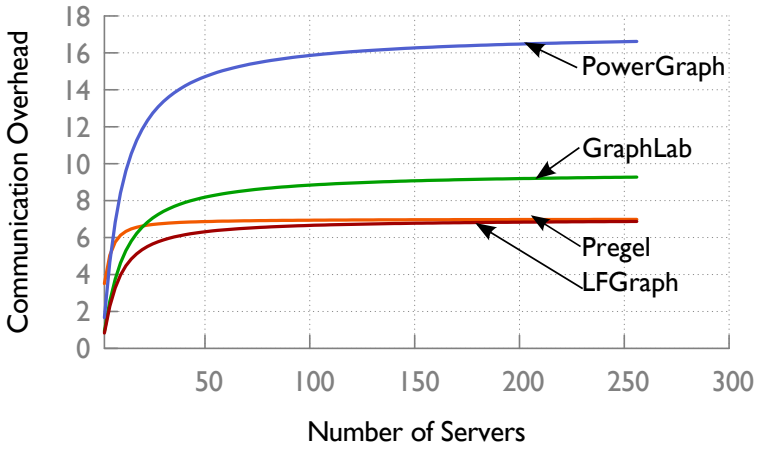
We calculate the equations of Section 2.4.1 for each graph by considering each of its vertices and the neighbors of those vertices. This denotes the communication overhead, i.e., number of values sent over the network per iteration per vertex. Figure 2.3 plots this quantity for different cluster sizes ranging from 2 to 256.



(a) Twitter



(b) UK-2007 Web Graph



(c) Amazon Recommendation Graph

Figure 2.3: Communication overhead for real-world graphs

First we observe that in all three plots, among all compared approaches, LFGraph has the lowest communication overhead. This is largely due to its fetch-once behavior.

Second, Pregel plateaus out quickly. In fact, the knee of its curve occurs (for each of the graphs) around the region where the x-axis value is in the ballpark of the graph's average out-degree. Up to that inflection point, there is a high probability that a randomly selected vertex will have neighbors on almost all the servers in the system. Beyond the knee, this probability drops.

Third, LFGraph's improvement over GraphLab is higher for the Twitter workload than for the other two workloads. This is because the in-neighbor and out-neighbor sets are more disjoint in the Twitter graph than they are in the UK-2007 and Amazon graphs.

Fourth, in Figure 2.3(c) (Amazon workload), when cluster size goes beyond 10, GraphLab's overhead is higher than Pregel's. This is because on an Amazon book webpage, there is a cap on the number of recommendations (out-neighbors). Thus out-degree is capped, but in-degree is unrestricted. Further, average value of $|D_{out}[v] \cup D_{in}[v]|$ is lower in the Amazon workload (9.58) than in Twitter (57.74) and UK-2007 (62.56). Finally, as the cluster size increases, GraphLab's communication overhead plateaus, with the knee at around the value of $|D_{out}[v] \cup D_{in}[v]|$ – this is because when $N \gg |D_{out}[v] \cup D_{in}[v]|$ in eq. 3, $C_{GL}(v) \approx |D_{out}[v] \cup D_{in}[v]|$. For Twitter and UK-2007, N is never large enough for GraphLab's overhead to reach its cap. Hence, GraphLab's overhead stays lower than Pregel's for those two workloads.

We conclude that in real-world directed graphs, LFGraph's hash-based partitioning and fetch-once communication suffices to achieve a lower communication overhead than existing approaches.

2.5 Load Balance in Real Graphs

We use synthetic power-law graphs and the three real-world graphs from Table 2.3 to analyze the computation balancing and communication balancing in LFGraph.

2.5.1 Computation Balance

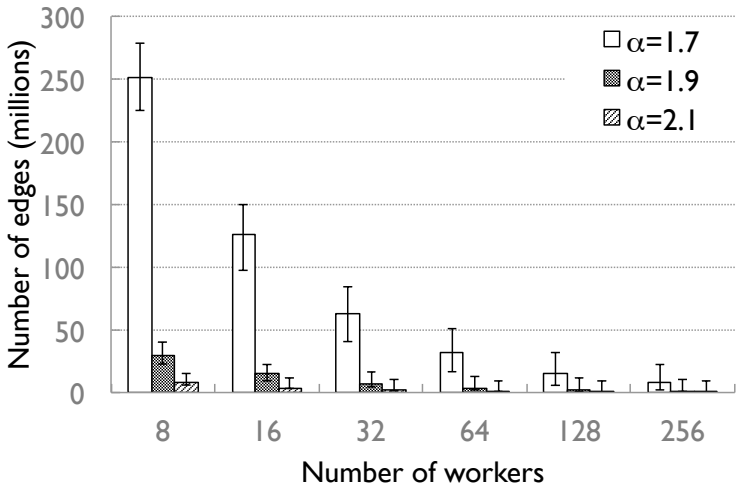


Figure 2.4: Computation overhead for synthetic power-law graphs

The completion time of an iteration is influenced by imbalance across computation workers. This is because tail workers that take longer than others will cause the entire iteration to finish later.

Prior works [26, 44, 40] have hypothesized that power-law graphs cause substantial computation imbalance, and since real-world graphs are similar to power-law graphs, they do too. In fact, the community has treated this hypothesis as the motivation for intelligent placement schemes, e.g., edge placement [26]. We now debunk this hypothesis by showing that when using random partitioning, while synthetic power-law graphs do cause computation imbalance, real-world power-law graphs in fact do not. The primary reason

is subtle differences between the in-degree distributions of these two types of graphs.

First we examine synthetic power-law graphs to see why they do exhibit computation imbalance. A power-law graph has degree d with probability proportional to $d^{-\alpha}$. Lower α means a denser graph with more high degree vertices. We created three synthetic graphs with 10 million vertices each with $\alpha = \{1.7, 1.9, 2.1\}$. We first selected the in-degree of each vertex using the power-law distribution. We then assigned vertices to servers using hash-based partitioning. For simplicity the rest of this section assumes only one computation worker per server.

In LFGraph the runtime of a computation worker during an iteration is proportional to the total number of in-edges processed by that worker. We thus use the latter as a measure for the load at a computation worker. Figure 2.4 plots, for different cluster sizes, LFGraph's average computation overhead along with error bars that show the maximum and minimum loaded workers. As expected, the average computation load falls inversely with increasing cluster size. However, the error bars show that all synthetic graphs suffer from computation imbalance. This is especially prominent in large clusters – with 256 servers, the maximum loaded worker is 35x slower than the average worker. In fact these slowest workers were the ones assigned the highest degree vertices.

Next, we repeat the same experiment on the three real-world power-law graphs from Table 2.3. Figure 2.5 depicts, for the three real-world graphs, average computation load along with error bars for maximum and minimum computation load. We see that unlike in Figure 2.4, the bars are much smaller here. In fact the maximum loaded worker is only 7% slower than the average worker.

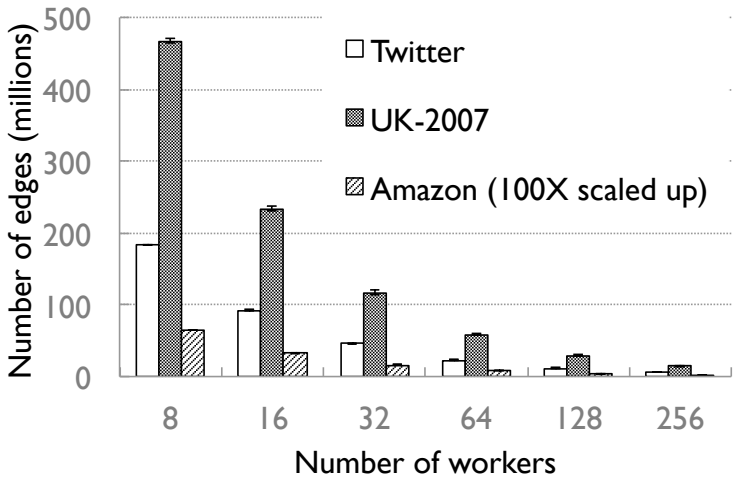


Figure 2.5: Computation overhead for real-world graphs

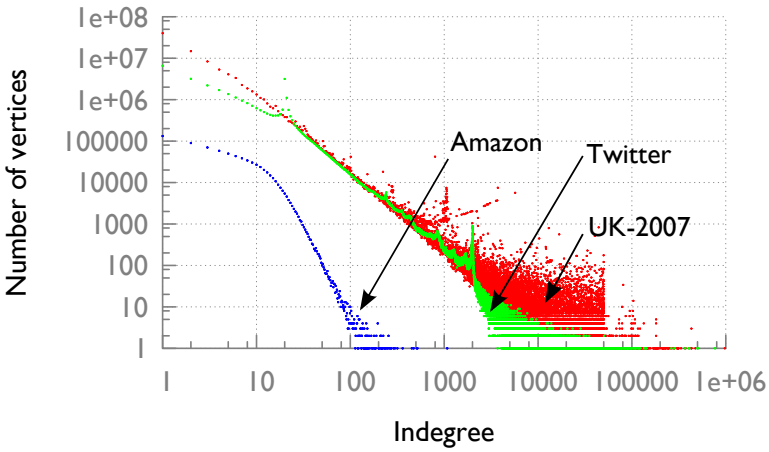


Figure 2.6: In-degree distribution for real-world graphs

The primary reason for this difference in behavior between synthetic and real-world power-law graphs is shown in Figure 2.5, which plots the in-degree distributions of the three real graphs. While synthetic power-law graphs have a straight tail in their log-log plot, each of the real power-law graphs has a *funnel* at its tail. Other real-world power-law graphs have also been shown to exhibit this pattern [11]. This indicates that in real-world power-law graphs, among the vertices with high degrees, there is substantial variability across the actual degrees – this is not the case in the idealized power-law graphs.

Since the number of such high degree vertices is far more than the number of servers, their resultant load balances out across servers. We conclude that hash-based partitioning suffices for power-law graphs, and that intelligent placement schemes will yield little benefit in practice.

2.5.2 Communication Balance

Communication imbalance is important because it can lead to increased processing time. During the communication phase each server receives vertex data from all other servers. Since the transfers are in parallel, if a server Si receives more data from Sj than from Sk, the overall data transfer time will increase even if the average communication overhead stays low.

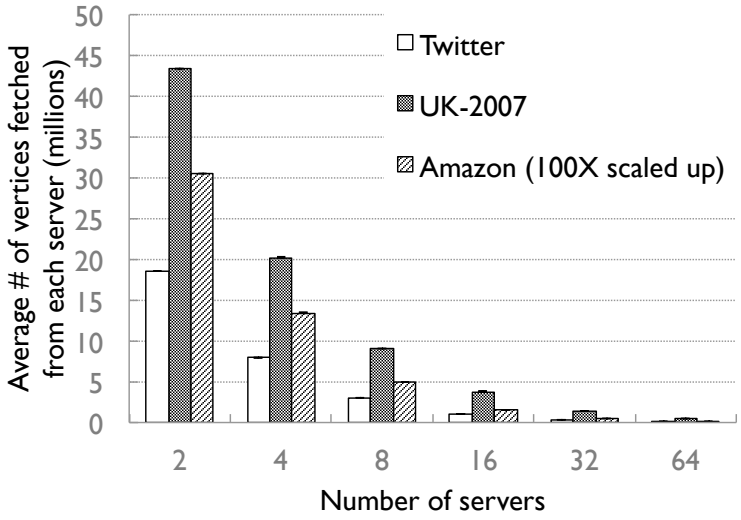


Figure 2.7: Communication overhead for real-world graphs

For the three real-world graphs we measure the vertex data transferred between every pair of servers. Figure 2.7 plots the average along with bars for maximum and minimum. The small bars indicate that LFGraph balances communication load well.

2.6 Fault Tolerance

We briefly discuss how LFGraph handles failures. The front-end receives heartbeats from JobServers. On detecting a failure the front-end replaces the failed JobServer with a new one and restarts the computation. Fault tolerance can also be achieved more efficiently in LFGraph. Note that in the communication phase vertices communicate their values to their neighbors. Therefore, the vertex values are already replicated in other servers. So, in case of a failure, computation can be restarted from the current iteration using these replicated values at a different JobServer.

2.7 Related Work

Single-Server Systems: Single-server graph processing systems are limited by memory and cannot process large graphs [45]. GraphChi [38] leverages the disk for large graph processing on a single server, making it slower than distributed frameworks. Grace [46] relies on machines with many cores and large RAM. It has two drawbacks – parallelizability is limited by the number of cores, and real-world large graphs cannot be stored in a single machine's memory.

Distributed Graph Processing Frameworks: Pregel has been the inspiration for several distributed graph processing systems, including Giraph [47], GoldenOrb [48], Phoebus [49], etc. These systems suffer from unscalability at small cluster sizes. For instance, we found that Giraph was unable to load a graph with 10M vertices and 1B edges on a 64 node cluster (16 GB memory each). Others reported similar experiences [26].

GraphLab [50, 37] and PowerGraph [26] also support asynchronous computations. Asynchronous models do not have barriers – so, fast workers do

not have to wait for slow workers. We performed experiments with these asynchronous variants but found that the runtime did not change much from the synchronous variants. Further, asynchrony makes it difficult to reason about and debug such programs.

Distributed-matrix models have been used for graph processing [51, 52]. These are harder to code in as they do not follow the more intuitive 'think like a vertex' paradigm (i.e., vertex programs) that Pregel, GraphLab, and PowerGraph do.

Finally, Piccolo [53] and Trinity [54] realize a distributed in-memory key-value store for graph processing. Trinity also supports online query processing on graphs and is known to outperform MPI-based implementations such as PBGL [55]. However, both Piccolo and Trinity require locking for concurrency control. We performed experiments with Trinity and observed that LFGraph achieves a 1.6x improvement.

General-purpose Data Processing Frameworks: General-purpose data processing frameworks such as MapReduce [56] and its variants [57], Spark [58], etc. can be used for graph analytics [59, 60, 61, 62]. However, they are not targeted at graph computations, thus they are difficult to program graph algorithms in due to the model mismatch. Further their performance is not competitive with graph processing frameworks [26].

Graph Databases: Graph databases such as FlockDB [25], InfiniteGraph [63], and Neo4j [64] are increasingly being used to store graph-structured data. Although these databases support efficient query and traversal on graph-structured data, they are not designed for graph analytics operations.

Performance Optimization: Various techniques have been designed to optimize performance of graph-based computations. These techniques include multilevel partitioning [65, 66, 67], greedy partitioning [68], join par-

titioning and replication strategies [69]. Based on our results, we believe that such complex partitioning schemes can be avoided while still improving performance. GPS [70] uses dynamic repartitioning schemes for runtime optimization. Mizan [44] uses dynamic load balancing for fast processing, Surfer [71] uses bandwidth-aware placement techniques to minimize cross-partition communication. These dynamic techniques can be applied orthogonally inside LFGraph.

3 BONDHU: SOCIAL NETWORK-AWARE DISK MANAGER FOR GRAPH DATABASES

Graph databases are increasingly being used to store networked data. Existing graph databases do not deploy disk layout techniques to improve data placement on disk and thus suffer in performance. In this chapter, we first present disk layout techniques that leverage small-world (i.e., community) structure in the underlying power-law graphs to make better placement decisions. Second, we build a layout manager called the Bondhu system that incorporates our techniques. We integrate Bondhu into the popular Neo4j graph database engine. The discussions in this chapter are based on one of the most prevalent small-world power-law graphs – Online Social Networks (OSNs). However, our techniques apply to any other graphs with similar properties.

The rest of the chapter is organized as follows. Section 3.1 presents the necessity of power-law aware techniques for the disk layout problem. Section 3.2 presents a formal definition of the disk layout problem. Section 3.3 discusses the disk layout algorithms which are at the core of the Bondhu system. Section 3.4 gives details of the prototype implementation of the Bondhu system in Neo4j. Section 3.5 presents three models for capturing user interactions in OSNs that we use in our experiments. Related works are presented in Section 3.6. We analyze the experimental results of our prototype implementation later in Chapter 5.

3.1 Motivation

The last few years have seen an unprecedented growth both in variety and in scale of Online Social Networks (OSN). This has led to the creation of graph databases for efficient OSN data storage. The OSNs stored by these databases are power-law in nature. In addition these graphs tend to be small world and exhibit unique structural properties such as strong community structure. This makes disk access patterns of OSN applications different from those of traditional applications. Our work is motivated by the observation that in order to improve disk access performance of OSN applications, it is critical to design techniques that take the structure of the graph into consideration. Although the discussion in this chapter is presented in the context of OSNs, the conclusions hold true for any other graphs with similar characteristics.

There have been several efforts to improve disk performance by careful data organization. The Fast File System improves disk performance by keeping related data blocks and their meta-data together [72]. Multimedia file systems use the organ pipe layout algorithm by tracking the popularity of the objects and keep the hottest object in cylinder zero and place successive cooler records to the left and right respectively [73, 74]. Others track block access patterns and try to place correlated blocks together on the disk [75, 76]. The Free Space File System makes use of the empty space of the disk to replicate blocks according to the observed access patterns [77].

The above approaches are suitable for traditional workloads, such as multimedia file systems, version control systems, and web servers. However, the access patterns in OSN applications are quite different from the above access patterns. This is due to many reasons, two of which we briefly discuss here.

In a multimedia system popular objects (movies, for example) are popular across all users. On the other hand, in an OSN scenario it is not the case that a few objects dominate globally. Rather, each user accesses her friends' information with a certain probability. Further, existing systems that track the access pattern of blocks and keep related blocks together are less likely to perform well due to the large scale of OSNs. Most of the OSNs consist of millions of users and thus tracking block level access patterns at that scale is not feasible.

Finding a good disk layout can be helpful in many ways. We mention a specific example here. A simple social network is stored in a graph database as blocks of profile information (name, address, phone, etc.) for users. When a user issues a query to obtain the name of all of her friends, the disk head has to seek the appropriate locations in the disk to read her friends' information. In the absence of a layout manager of the graph database, related users' data will be scattered all over the disk and hence the disk head movement is increased. On the other hand, a good layout might reduce disk head movement by keeping *related* users' data close by on the disk. This would translate to faster response time in answering the queries.

We motivate this further by presenting a visualization of disk block access patterns of a sample OSN application in Figure 3.1. We use the Facebook New Orleans network graph [78] to build a sample OSN application using the Neo4j graph database [64] (more details are in Section 5). For each user in the social graph, we create a *node* in Neo4j. Then we store a 400KB data block (*property* in Neo4j) for each user. Next we write an automated script that logs into the system as a random user and retrieves the data blocks for all of her friends. This is identical to the 'list all friends' action in an OSN. We trace the disk blocks accessed by each request using the *blktrace*

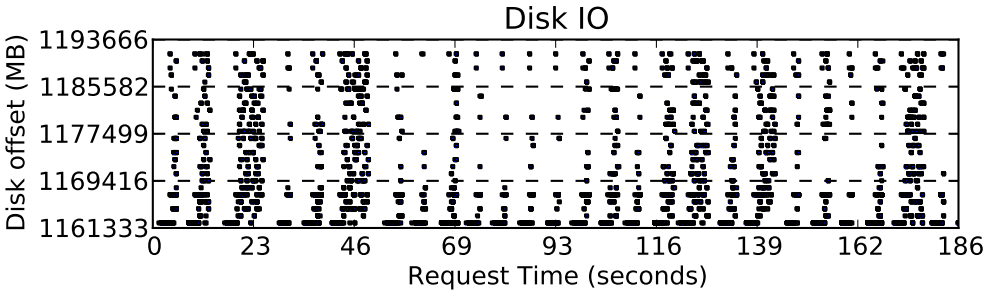


Figure 3.1: Blocks accessed in Neo4j for a 'list friend' query

tool [79] and use the *seekwatcher* tool [80] to visualize the disk block access over time. A dot in Figure 3.1 depicts a block access at a particular location on the disk at a particular time. We observe from the figure that block accesses are scattered all over the database. This effect is prominent when the queries are issued by users with many friends (around 23 and 46 seconds, for example). Therefore, the disk head has to move a lot to answer this query, which leads to a high response time. Later in Figure 5.1, we show how social network-aware disk placement performs better for the above workload.

We believe that graph-aware data organization scheme can improve disk access performance because it changes the random and scattered movement pattern of the disk head to one which is semi-sequential and confined within smaller regions. To examine how bad the random access performance of a disk is compared to the sequential access performance, we measure disk throughput under both access patterns using the fio benchmarking tool on 3 different hard disks: a 4 year old desktop hard disk (SEGATE), a 2 year old datacenter hard disk (HP), and a new desktop hard disk (SAMSUNG). The results are presented in Figure 3.2. In all the three disks the random access performance is more than two orders of magnitude worse than the sequential access performance. Therefore, a layout that takes the disk access pattern into account and organizes the data accordingly can improve performance

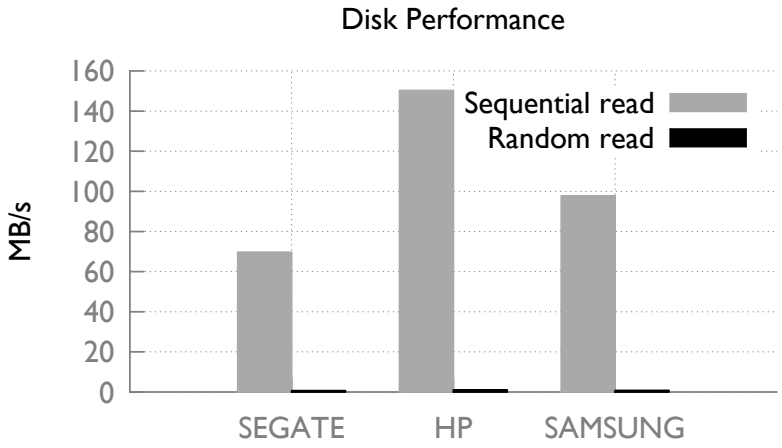


Figure 3.2: Sequential vs. random read for 3 disk types

significantly.

While solid state disks (SSDs) are becoming increasingly viable alternatives to disks, we believe that disks will not go away. For instance, due to write lifetime issues with SSDs, hard disks are often used as a cache for SSDs [81]. In addition, disks are likely to be cheaper than SSDs per byte for several years.

Motivated by the above discussion, in this chapter, we present the design and implementation of the Bondhu System which leverages the power-law small world social network graph to intelligently layout data on disk. The layout schemes of the Bondhu system improves the disk performance because of three reasons: i) when the user block sizes are small, the data fetched in a single seek contains multiple friends' data, lowering the number of seeks; ii) the disk arm movement is reduced as related data are clustered together – this leads to a lower seek distance (time); iii) rotational latency is improved since the disk has to rotate less to reach the appropriate location for fetching data.

Concretely, we make the following contributions in this chapter:

- We present a novel framework for disk layout algorithms based on community detection in a social (power-law small world) graph. First, we detect the communities within a graph. Then, we produce the layout by running a greedy heuristic within and across the communities. To the best of our knowledge, Bondhu is the first system that leverages the underlying power-law graph for efficient data layout in disks.

- We implement our solution into Neo4j, which is a widely used open source graph database. We show through experimentation that the Bondhu system is able to improve response time by as much as 48% when compared to the default layout policy implemented by the file system.

- Our experiments, using Facebook traces, show that while taking the graph structure into account helps make better placement decisions, taking the user access patterns into account yields low additional benefit.

3.2 Problem Definition

Consider N users: $V = \{V_1, V_2, \ldots, V_N\}$, and N consecutive locations on disk denoted by: $L = \{L_1, L_2, \ldots, L_N\}$. Now, consider a function $\delta(V_i, V_j)$ representing the social network.

$$\delta(V_i, V_j) = \begin{cases} 0 & \text{if } V_i,\ V_j \text{ are not friends} \\ 1 & \text{if } V_i,\ V_j \text{ are friends} \end{cases}$$

We assume that relationships are symmetric, i.e., $\delta(V_i, V_j) = \delta(V_j, V_i)$ for all (i, j). Define $loc(.)$ to be a one-to-one function which denotes a particular

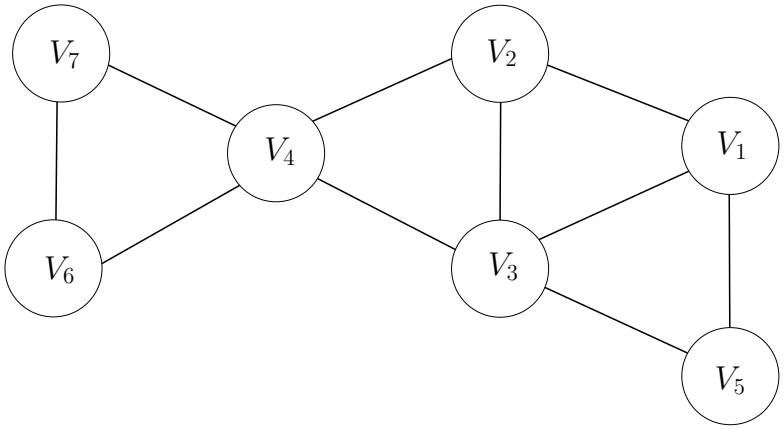


Figure 3.3: A sample social graph

'layout', i.e., location arrangement, $loc : V \rightarrow L$. There are $N!$ possible $loc(.)$ functions. Further, the *cost* of a layout from the perspective of a particular user V_i is given by the sum of the difference of the disk locations between the user and all of her friends. The lower the *cost*, the lower the seek distance, and the better the response time. Therefore,

$$cost_i = \sum_{j=1}^{N} [|loc(V_i) - loc(V_j)| * \delta(V_i, V_j)] \tag{3.1}$$

Therefore, the total cost of a layout is:

$$\begin{aligned} cost &= \frac{\sum_{i=1}^{N} cost_i}{2} \\ &= \frac{\sum_{i=1}^{N} \sum_{j=1}^{N} \{|loc(V_i) - loc(V_j)| * \delta(V_i, V_j)\}}{2} \end{aligned} \tag{3.2}$$

The lower the cost of a layout, the closer the friends of a user are located on the disk. This speeds up common operations like friend listing, publish-subscribe of wall-posts, etc. Therefore, our goal is to find the layout with the minimum cost.

Table 3.1: Cost of the linear layout

Location	L_1	L_2	L_3	L_4	L_5	L_6	L_7
User	V_1	V_2	V_3	V_4	V_5	V_6	V_7
V_1	-	1	2	0	4	0	0
V_2	-	-	1	2	0	0	0
V_3	-	-	-	1	2	0	0
V_4	-	-	-	-	0	2	3
V_5	-	-	-	-	-	0	0
V_6	-	-	-	-	-	-	0
V_7	-	-	-	-	-	-	-

Table 3.2: Cost of one of the optimal layouts

Location	L_1	L_2	L_3	L_4	L_5	L_6	L_7
User	V_5	V_1	V_3	V_2	V_4	V_6	V_7
V_5	-	1	2	0	0	0	0
V_1	-	-	1	2	0	0	0
V_3	-	-	-	1	2	0	0
V_2	-	-	-	-	1	0	0
V_4	-	-	-	-	-	1	2
V_6	-	-	-	-	-	-	1
V_7	-	-	-	-	-	-	-

We illustrate the problem with the help of the sample social graph in Figure 3.3 with 7 users. Consider the linear layout in Table 3.1: V_1 at L_1, V_2 at L_2, and so on. The users are arranged in the rows and columns according to their layout. An entry (V_i, V_j) in the table is non-zero if there is a link between V_i and V_j in the graph (in other words if V_i and V_j are friends), otherwise it is 0. The non-zero value is the absolute value of the difference of the locations of V_i and V_j (i.e., it is the cost as defined before). Adding up all the values we get the cost of the layout = 18. However, this is not optimal. We present one of the optimal layouts in Table 3.2 with cost = 14.

This min-cost social network embedding problem is a variant of the Minimum Linear Arrangement problem, which is known to be NP-hard [82]. The best known heuristic to solve this problem is Simulated Annealing, which itself is computationally infeasible for large graphs [83].

In this chapter, we first propose a fast multi-level heuristic to solve this problem, which can handle graphs with millions of nodes. The Bondhu system uses this algorithm to obtain disk layout.

Second, we solve the weighted version of this problem. We use weighted graphs to capture user interactions in the social network. A high edge weight between two users implies that they are more likely to access each other's data and so they should be close by in the disk layout. We use the function $\delta(V_i, V_j)$ to capture the edge weight (w). Thus,

$$\delta(V_i, V_j) = \begin{cases} 0 & \text{if } V_i, V_j \text{ are not friends} \\ w & \text{if } V_i, V_j \text{ are friends} \end{cases}$$

We make one final point about disk geometries before we present our techniques. While disk geometries are often proprietary, manufacturers do present a logical abstraction of the disk which is known as the Logical Block Addressing (LBA) scheme. It is a simple linear addressing scheme where blocks are addressed by an integer index starting from 0. The LBA scheme is essentially a one-dimensional representation of the complex physical geometry of the disk. Disk manufacturers ensure that accessing consecutive blocks in the LBA space is similar to accessing consecutive blocks in the physical geometry. Experimental results [84, 77] also support this claim. Therefore, we use this simple one-dimensional model of the disk for data layout.

3.3 Disk Layout Algorithm

In this section we present the disk layout algorithm of the Bondhu system. At first we present the intuition behind our proposed algorithm and then explain it in detail in the following subsections.

3.3.1 Overview

OSNs are power-law graphs which are small world in nature. These are known to exhibit strong community structure. Therefore, we adopt an approach to disk layout algorithms for OSN applications that take the community structure into account. This has multiple benefits. First, the problem space is reduced. So, while making a disk placement decision inside a community we can consider only the members of that community. Second, a bad placement choice will have relatively less impact since the worst possible placement will likely be limited by the community boundary. Third, we can use the existing community detection techniques that are known to find good quality communities in a social graph.

Motivated by these observations, we present the layout algorithm of the Bondhu system. Figure 3.4 illustrates our approach. The algorithm consists of three modules: i) Community Detection: using existing community detection techniques, we divide the social graph into several communities, ii) Intra-Community Layout: using a greedy heuristic we find a layout for the users within the communities, and iii) Inter-Community Layout: we organize the different communities on the disk by considering inter-community tie strength. These three parts of the framework are discussed below.

3.3.2 Community Detection

The goal of the community detection module is to organize the users of the social graph into clusters, so that many edges connect users within the same cluster and relatively few edges connect users in different clusters. The community detection module makes use of existing techniques for graph partitioning and modularity optimization. We select these two algorithm classes

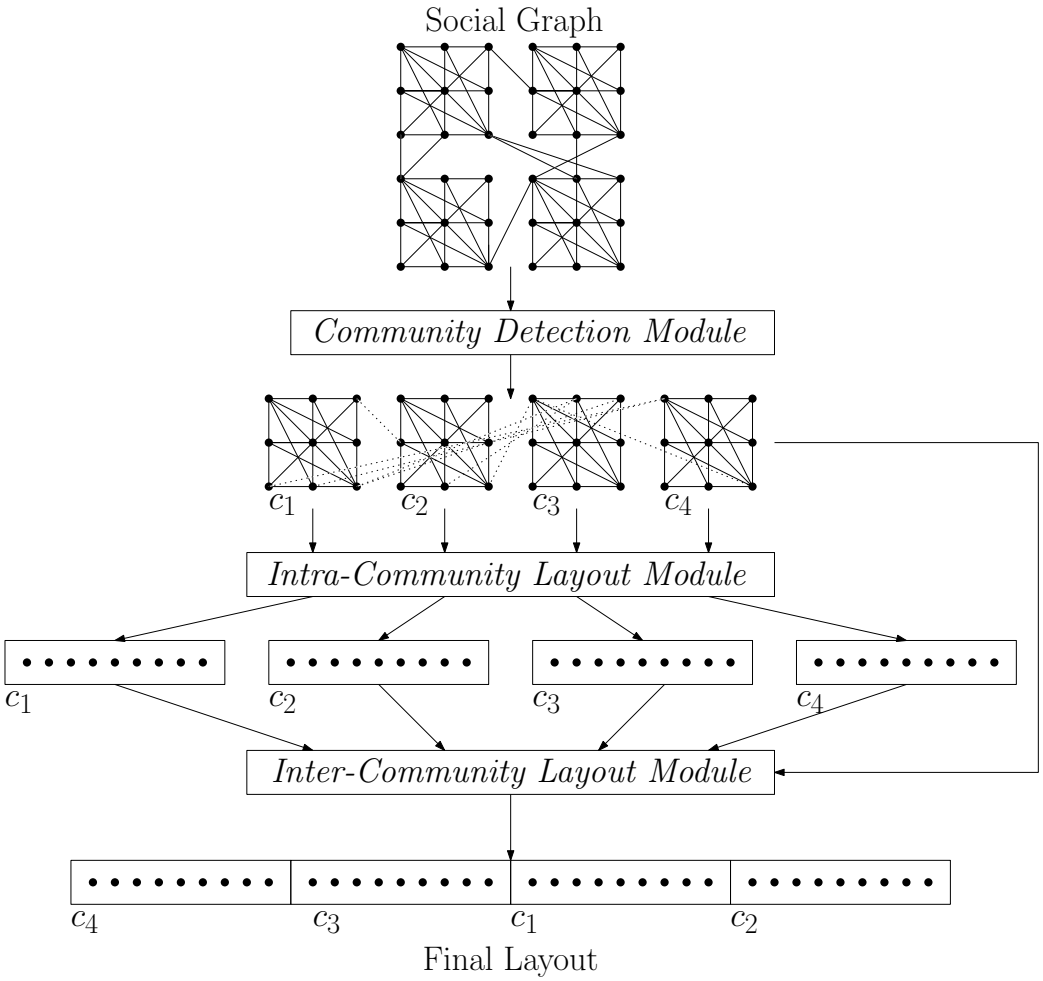


Figure 3.4: Overview of the Bondhu system's approach

because: i) they operate on graphs with large number of vertices, ii) they are known to produce good clusterings, and iii) they are fast, i.e., can find communities on graphs containing millions of nodes within seconds. We briefly discuss the algorithms here.

Graph Partition Driven Community Detection

Our graph partition driven community detection algorithm (ParCom) is based on the multilevel k-way hypergraph partitioning scheme of [67, 85]. The goal of ParCom is to partition the social graph into k equal subsets such that the edge-cut is minimized. This is equivalent to minimizing the

number of friends in other partitions. ParCom works as follows. First, the social graph is coarsened down to a small number of vertices. In this phase a sequence of smaller graphs is constructed from the original graph by collapsing vertices together using the heavy-edge matching (HEM) technique. The weights of the edges are also recalculated. Then this smaller graph is divided into k-parts using recursive bi-section scheme. Finally the partitions are uncoarsened back to the original graph in steps and at each step the partitions are refined using local refinement heuristics. Fore more details, see [67, 85].

Modularity Optimization Driven Community Detection

Our modularity optimization driven community detection algorithm (ModCom) is based on [86]. It is able to detect good quality communities in large networks (118 million nodes).

The modularity of a partition is a scalar value between -1 and 1 that measures the density of intra-community links as compared to inter-community links. More specifically, modularity is defined as the fraction of edges that fall within the communities minus the expected value of the fraction if the edges were randomly distributed (by preserving node degrees). Formally, it is defined as:

$$Q = \frac{1}{2M} \sum_{V_i, V_j} \left[\delta(V_i, V_j) - \frac{k_i k_j}{2M} \right] \sigma(c_i, c_j) \tag{3.3}$$

Here, M = number of edges, $\delta(V_i, V_j)$ = weight of the edge between user V_i and V_j, k_i = degree of user V_i (sum of the weights of the links/edges connected to user V_i), c_i = community of user V_i, and $\sigma(c_i, c_j) = 1$, if $c_i = c_j$, and 0 otherwise.

ModCom works in two phases. In the first phase users are arranged in

a random order and each of the users is assigned to a different community. Then for each user V_i the gain in modularity is calculated by removing it from its own community and by assigning it to any of its friends' communities. User V_i is then moved to its friend V_j's community, for which the modularity gain is maximum. In case of no modularity gain, V_i stays in its original community. This first phase is repeated iteratively for all of the users until no further gain in modularity is possible. In the second phase, a new graph consisting of the communities obtained in the first phase is created. Note that the edge weights are recalculated for this phase. After this, the first phase is run again and the process is continued until no further changes are possible. For more details, see [86].

It is important to note that the difference between ModCom and ParCom is that in ModCom the number of communities cannot be controlled explicitly as it can be in ParCom. This affects our later experimentation.

3.3.3 Intra-Community Layout

Next, the intra-community layout module takes as input the communities that are produced by the community detection module. For each community it creates a layout for the users within that community. We use a greedy heuristic to find a layout for each of the communities. The heuristic works as follows. We start with the most popular user, i.e., the user with the highest edge degree (=sum of link weights) and place that user in the middle of the disk layout. Next, among all the friends of that user we choose the one that is connected to the user with the heaviest edge. This is to ensure that if two users are strongly connected, they should be placed close by on the layout. In case of a tie, we choose the friend with the higher edge degree

(the more popular friend). Intuitively, by adding a popular user early, we provide more choice for the greedy algorithm. We place the friend to the left of the already placed user on the layout. We then create a modified graph by merging the user and her friend. The edges connected to these two users are now connected to the combined node. In case of a common friend of the two users, we assign the weight of the edge between the combined node and the common friend as the sum of the individual edge weights.

Next, among all the friends of this combined node we choose the one that is connected to it with the heaviest edge and place it on the right. We repeat the above steps iteratively by placing the friends to the left and to the right of the already placed users alternatively.

The different components of the algorithm are presented in Algorithm 3.1 (layout algorithm), Algorithm 3.2 (finding the maximally connected friend), and Algorithm 3.3 (creation of the combined node).

3.3.4 Inter-Community Layout

Our third component is the inter-community layout module. It takes as input: i) the communities produced by the community detection module, and ii) the layout produced within each community by the intra-community layout module. The goal of this component is to create a layout of the communities. This enables us to capture the relationships among different communities. For example, if a community c_i is strongly connected to another community c_j, these two should be placed close by on the disk – this reasoning is similar to the one used for the intra-community layout module.

To create the inter-community layout, we create a graph using the different communities as vertices. The weight of the edge between community

Algorithm 3.1 Calculate Layout L on Graph $G = (V, E(w))$

$enum\{RIGHT = 1, LEFT = 2\}$
$left \leftarrow right \leftarrow \frac{(N+1)}{2}$
$V_c \leftarrow \emptyset$
$direction \leftarrow RIGHT$
//continue until we combine all the nodes
while $size(G) > 1$ **do**
    //find the friend who is maximally connected to V_c
    //in case of $V_c = \emptyset$, return the node with the highest edge degree
    $V_i \leftarrow max\_connected(V_c)$
    //combine V_c and V_i to create a new graph with recalculated edge weights
    $(G, V_c) \leftarrow combine(V_c, V_i, G)$
    //alternate between $left$ and $right$ to place V_i
    if $direction = LEFT$ **then**
        $L_{left} \leftarrow V_i$
        $right \leftarrow right + 1$
        $direction \leftarrow RIGHT$
    else
        $L_{right} \leftarrow V_i$
        $left \leftarrow left - 1$
        $direction \leftarrow LEFT$
    end if
end while

Algorithm 3.2 *max_connected*(V_c)

if [**then**initial state]$V_c = \emptyset$
  //return the one with the highest edge degree
  $V_s \leftarrow V_i \,|\, edgeDegree(V_i) \geq edgeDegree(V_j)$,
    $\forall V_i \in V, \forall V_j \in V, V_i \neq V_j$
  if $size(V_s) > 1$ **then**
    //return a random one in case of tie **return** $random(V_i) \,|\, V_i \in V_r$
  elsereturn V_s
  end if
else[normal case operation]
  //select the friend connected to the heaviest edge of V_c
  $V_s \leftarrow V_i \,|\, edgeWeight(V_c, V_i) \geq edgeWeight(V_c, V_j)$,
    $\forall V_i \in friend(V_c), \forall V_j \in friend(V_c), V_i \neq V_j$
  if [**then**there is a tie]$size(V_s) > 1$
    //select the one with the highest edge degree
    $V_r \leftarrow V_i \,|\, edgeDegree(V_i) \geq edgeDegree(V_j)$,
      $\forall V_i \in V_s, \forall V_j \in V_s, V_i \neq V_j$
    if [**then**there is a tie again]$size(V_r) > 1$
      //return a random one from the list **return** $random(V_i) \,|\, V_i \in V_r$
    elsereturn V_r
    end if
  elsereturn V_s
  end if
end if

Algorithm 3.3 $combine(V_c, V_i, G = (V, E(w)))$

```
//create a new node by joining V_c & V_i
V'_c ← createNode(V_c, V_i)
//add the new node to the set of vertices
V ← V ∪ V'_c
    //start by deleting the edge between V_c & V_i
deleteEdge(V_c, V_i)
for all F ∈ friend(V_c) do
    w ← edgeWeight(V_c, F)
    //delete edges between V_c & its friends
    deleteEdge(V_c, F)
    //add edges between the new node & V_c's friends
    addEdge(V'_c, F, w)
end for
for all F ∈ friend(V_i) do
    w ← edgeWeight(V_i, F)
    //delete edges between V_i & its friends
    deleteEdge(V_i, F)
    //in case of a common friend of V_c & V_i, we already created an edge
    if isEdge(V'_c, F) then
        w' ← edgeWeight(V'_c, F)
        //increase the weight of the already created edge
        setEdgeWeight(V'_c, F, w + w')
    else[otherwise create a new edge]
        addEdge(V'_c, F, w)
    end if
end for
V ← V − V_c − V_i //delete V_c & V_i from the set of vertices return (G, V'_c)
```

c_i and community c_j is calculated as the sum of the weights of the edges from the members of community c_i to the members of community c_j. After creating the community graph we run the same iterative algorithm as the intra-community layout module to find a layout of the communities.

When this is done, we expand the layout within each community, which was previously obtained from the intra-community layout module. This gives us the final disk layout containing all the users of the social graph.

Example: We present a working example of our techniques in Figure 3.5. This is the same graph as shown in Figure 3.3. First, the community detection module splits the graph into two communities: c_1=$\{V_4, V_6, V_7\}$ and c_2=$\{V_1, V_2, V_3, V_5\}$. Then, the intra-community module finds a layout for both of them separately. Let us examine the steps taken by the module for c_2. Here, the first user to be chosen can be either V_3 or V_1, since both of them have the highest edge weight (=3). The algorithm chooses V_3 at random and places it in the middle of the layout. Next, the algorithm considers V_1, V_2, and V_5 (highest edge weight connected to V_3=1). V_1 is chosen since it is the most popular of all (edge degree=3). V_3 and V_1 are combined to $V_{3,1}$ and a new graph is constructed. Now, the algorithm considers V_2 and V_5 (highest edge weight to $V_{3,1}$=2), and V_5 is chosen at random (both V_2 and V_5 are equally popular). $V_{3,1}$ and V_5 are combined to obtain $V_{3,1,5}$, which leaves the algorithm with the last user (V_2) to be placed. The final layout produced for c_2 is: $\{V_2, V_1, V_3, V_5\}$. Likewise, the layout produced for c_1 is: $\{V_7, V_6, V_4\}$. The steps for the inter-community layout module is trivial, since we only have two communities in this example. So, the final layout produced is either $\{c_2, c_1\}$=$\{V_2, V_1, V_3, V_5, V_7, V_6, V_4\}$, or $\{c_1, c_2\}$=$\{V_7, V_6, V_4, V_2, V_1, V_3, V_5\}$ depending on which community is chosen first by the inter-community layout module.

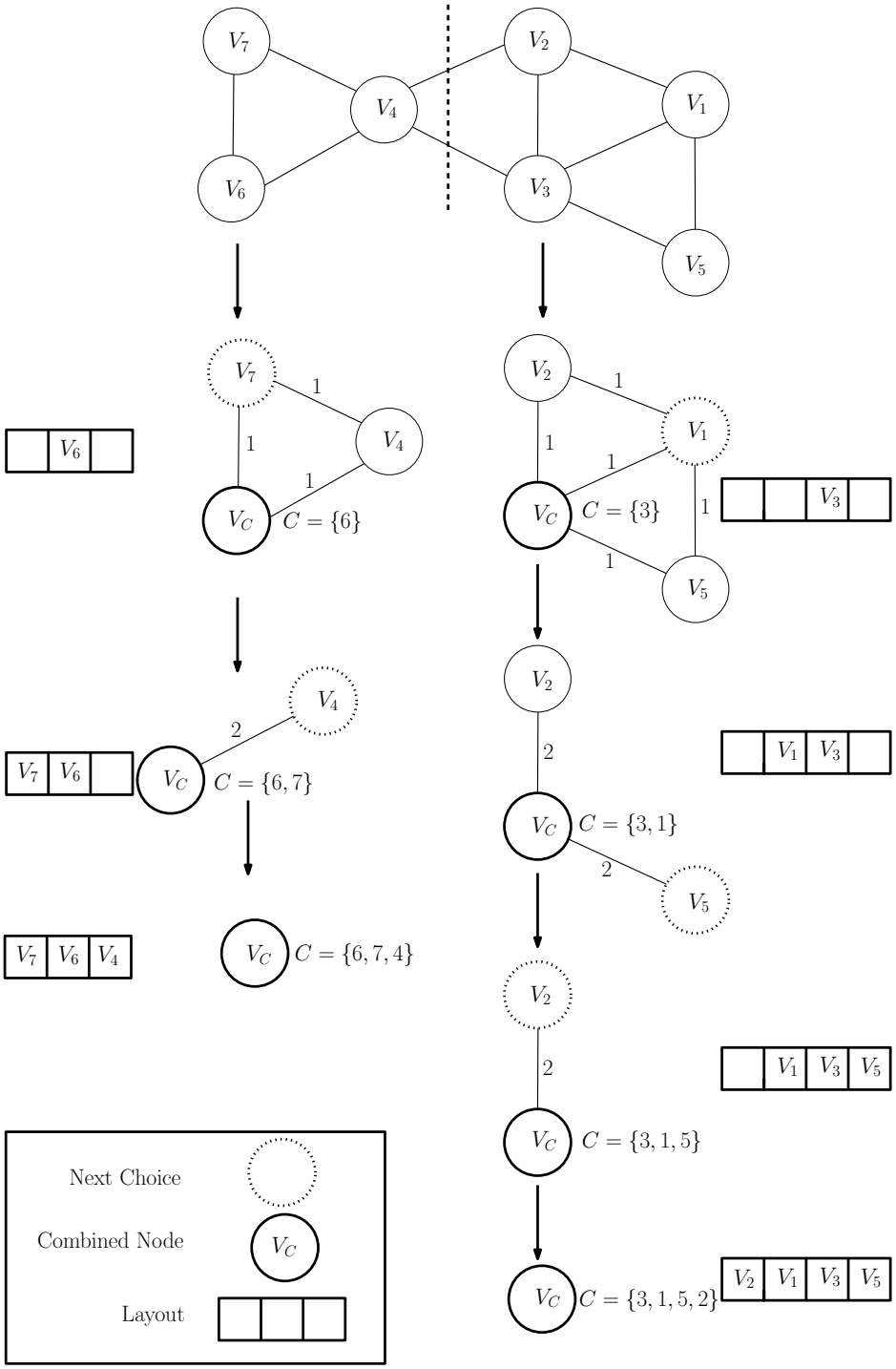


Figure 3.5: Working example

3.4 Implementation

We implement the Bondhu system as a layout manager for the Neo4j [64] graph database. Neo4j is a very popular and widely used graph database. It

is suitable for building OSN applications as it offers a graph-oriented model for data representation. A Neo4j graph consists of nodes, relationships, and properties. Properties are mapping from a string key to a value and can be associated with both nodes and relationships. The part of the Neo4j storage engine that stores properties is known as the PropertyStore.

We modify the PropertyStore of Neo4j so that the records are organized by the layout algorithms of the Bondhu system. Note that we rely on the native file system so our layout decisions are propagated to the disk block level, i.e., the modified PropertyStore database produced by the Bondhu system is stored sequentially on the disk. Therefore, we start with an empty disk and verify with the *davl* [87] tool that the database file is stored sequentially on the disk at the block level.

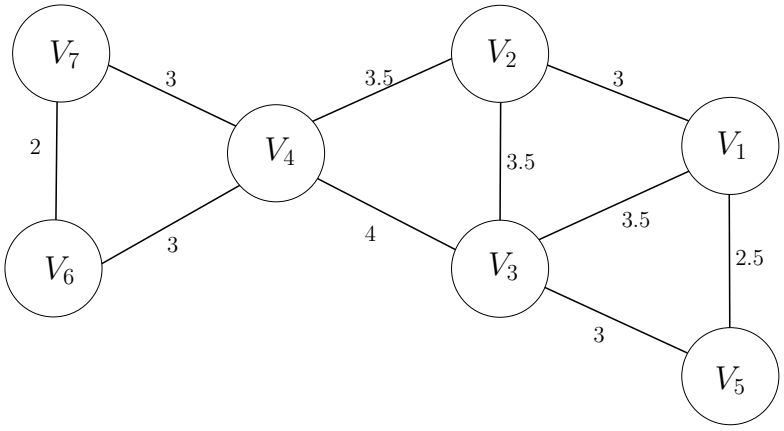
Our implementation of the Bondhu system is in Java. The community detection module makes use of the METIS library [88] for the ParCom algorithm.

3.5 Modeling the Social Network

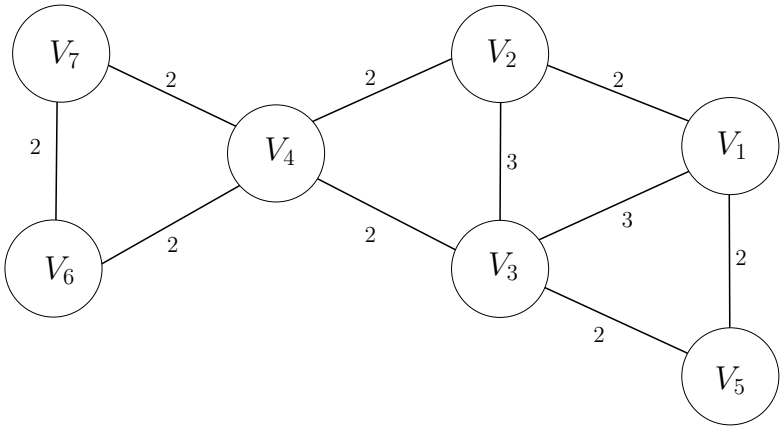
In this section we present three models to capture user interaction in a social network. We use these models to evaluate our disk layout techniques later in Chapter 5. These models vary in the way they assign weights to the edges between users.

3.5.1 Uniform Model

The uniform model is the simplest of the three models. In this model we assign equal weight (=1) to all social network edges. In other words, according to this model a user is equally likely to access any of her friends' informa-



(a) Preferential Model



(b) Overlap Model

Figure 3.6: Modeling the social network

tion. Listing the name of all of the friends of a given user can be viewed as an example of this model.

3.5.2 Preferential Model

The preferential model is motivated from the observation that a user with large number of friends is likely to be more active than a user with fewer friends, e.g., make more status updates, post more frequently, etc. In other words, a user with a larger number of friends is more active than a user with fewer number of friends. While browsing, a user is more likely to access the information of the more active friends.

To capture this type of interaction, the weight of the edge (V_i, V_j) should be proportional to the edge degree of V_j. Note that this metric is not symmetric, i.e., if V_j has a higher edge degree than V_i, then the weight of (V_i, V_j) is higher than the weight of (V_j, V_i). On the other hand, disk locality is symmetric in nature and to capture that our social graph models are undirected. Therefore, according to the preferential model the weight of the edge (V_i, V_j) is set to $[edgeDegree(V_i) + edgeDegree(V_j)]/2$. In Figure 3.6(a) we assign the edge weights according to the preferential model.

3.5.3 Overlap Model

The overlap model is motivated by the following observation: two users with a large number of common friends are more likely to share common interests than two users with fewer number of common friends. Therefore, the two users with the larger number of common friends are more likely to access each other's information. In other words, if user V_i has p common friends with V_j and q common friends with V_k, and if $(p > q)$, then V_i is more likely

to access V_j's data than V_k's data.

To assign the weight of the edge (V_i, V_j) according to the overlap model, we calculate the number of common friends c between V_i and V_j and set the edge weight as $(c + 1)$. We add a 1 to make sure that we do not assign a 0 weight to the edge (V_i, V_j) in case of no common friends, since an edge weight of 0 indicates there is no edge at all. In Figure 3.6(b), we assign the edge weights according to the overlap model.

3.6 Related Work

Data organization techniques for improving disk performance broadly fall into two categories: i) access pattern-oblivious, and ii) access pattern-aware. Access pattern-oblivious techniques include placing data and meta-data together as in the Fast File System and its variants [72, 89, 90], writing data sequentially to contiguous disk segments as in the Log-structured File System [91], and explicitly grouping small files together on disk as in C-FFS [92]. Access pattern-aware techniques can be further categorized into three types depending on the level of abstraction they work at: i) cylinder level [93, 94, 95], ii) block level [96, 76, 77, 75, 97, 98], and iii) file system level [99].

The position of Bondhu is in the middle of these two extremes. On one hand, it is not access pattern-oblivious in the sense that it captures the community structure of the social network. On the other hand, it is not completely access pattern-aware in the sense that it does not make placement decisions based on the real traffic between users.

Aside from data organization, disk performance can be significantly improved using intelligent prefetching and caching techniques [76, 100]. C-

Miner [76], for example, uses data mining techniques to learn the block access patterns and uses that information to make prefetching and cache replacement decisions. The Bondhu system can be extended to make social network-aware prefetching decisions, which remains as one of our future works.

With the recent growth of OSNs, many focused on partitioning the social graph to make OSN applications scalable [101, 102, 69]. SPAR [69], for example, uses partitioning and replication techniques to reduce network traffic across servers. Bondhu, on the other hand uses partitioning and community detection techniques for disk performance improvement. An excellent survey on existing community detection techniques is available at [103].

4 EXPERIMENTAL EVALUATION OF LFGRAPH

In this chapter, we experimentally evaluate our LFGraph system (implemented in C++). We evaluate LFGraph from the viewpoint of runtime, memory footprint, as well as overhead and balancing w.r.t. both computation and communication. We show that our power-law-aware publish-subscribe mechanism lowers communication overhead significantly compared to existing systems. In addition, we show that LFGraph is able to achieve communication and computation load balance by using our hash-based placement scheme. This suggests that LFGraph successfully exploits the variability in the degree distribution of the high degree vertices of power-law graphs. These techniques help LFGraph perform better than existing graph analytics frameworks.

4.1 Experimental Setup

We compare LFGraph against the best-known graph processing system, i.e., PowerGraph [26], using its open-source version 2.1 [104].[1]

PowerGraph [104, 26] offers three partitioning schemes. In increasing order of complexity, these variants are: i) Random, ii) Batch (greedy partitioning without global coordination), and iii) Oblivious (greedy partitioning with global coordination).

Our target cluster is Emulab and our setup consists of 32 servers each with

[1] Although GraphLab has lower communication overhead, its implementation is slower than PowerGraph.

a quad-core Intel Xeon E5530, 12 GB of RAM, and connected via a 1 GigE network. Due to brevity, we present results from only: 1) the Twitter graph containing 41M vertices and 1.4B edges (Table 2.3), and 2) larger synthetic graphs with log-normal degree distribution, containing 1B vertices and up to 128B edges. Note that although some benchmarks such as Graph500 [105] target much larger graphs, they are intended for rating supercomputer systems. In addition, the vertex to server ratio is much lower in the Graph 500 benchmark compared to real-world analytics jobs. So, we focus on scenarios which conform to real-world settings.

We study three main benchmarks: i) PageRank, ii) Single-Source Shortest Path (SSSP), and iii) Triangle Counting. These applications are chosen because they exhibit different computation and communication patterns: in PageRank all vertices are active in all iterations, while in SSSP the number of active vertices rises in early iterations and falls in later iterations. Thus PageRank is more communication-intensive while SSSP exhibits communication heterogeneity across iterations. However, in both PageRank and SSSP the values maintained by vertices are of fixed size. So, we examine LFGraph's behavior under the Triangle Counting benchmark. In addition to the variable-sized values, the Triangle Counting benchmark operates on an undirected graph. So, it helps us to better understand LFGraph's performance under bidirectional information flows.

We summarize our key conclusions:

- For communication-heavy analytics such as PageRank, when including the partitioning overhead, LFGraph exhibits 5x to 380x improvement in runtime compared to PowerGraph, while lowering memory footprint by 8x to 12x.

- When ignoring the partitioning overhead, LFGraph's per-iteration runtime is 2x faster than the best PowerGraph variant.

- Intelligent partitioning is prohibitive at most cluster sizes. In a small cluster, distributed graph processing is compute-heavy, thus intelligent partitioning (e.g., in PowerGraph) has little effect. In a large cluster, intelligent partitioning can speed up iterations, however the partitioning cost itself increases with cluster size and contributes sizably to runtime.

- LFGraph's hash-based placement scheme achieves both computation and communication balance across workers, and lowers overall runtime.

4.2 PageRank Benchmark

4.2.1 Runtime

We ran the PageRank benchmark with 10 iterations[2] on the Twitter graph. Figure 4.1 compares LFGraph against the three PowerGraph variants. This plot depicts runtime *ignoring the partitioning iteration* for PowerGraph's Oblivious and Batch variants. Each datapoint is the median over 5 trials.

The reader will notice missing data points for PowerGraph at cluster sizes of 4 servers and fewer. This is because PowerGraph could not load the graph at these small cluster sizes – this is explained by the fact that it stores both in-links and out-links for each vertex, as well as book-keeping information, e.g., mirror locations.

Among the PowerGraph variants, random partitioning is the slowest compared to the intelligent partitioning approaches – this is as expected, since partitioning makes iterations faster. However, LFGraph is 2x faster than the

[2]Our conclusions hold even with larger number of iterations.

best PowerGraph variant. Thus, even on a per-iteration basis, LFGraph's one-pass compute and fetch-once behavior yields more benefit than PowerGraph's intelligent partitioning.

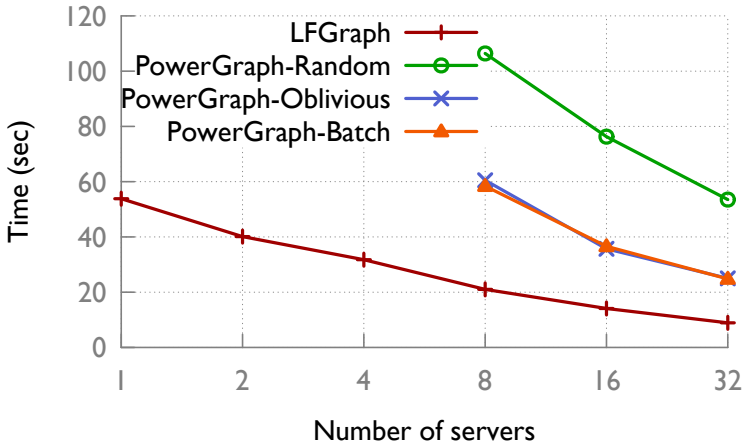


Figure 4.1: PageRank runtime comparison for Twitter graph (10 iterations), ignoring partition time.

Next, Figure 4.2 includes the partitioning overhead in the runtime, and shows runtime improvement of LFGraph. In a small cluster with 8 servers, LFGraph is between 4x to 100x faster than the PowerGraph variants. In a large cluster with 32 servers the improvement grows to 5x–380x. The improvement is the most over the intelligent partitioning variants of PowerGraph because LFGraph avoids expensive partitioning.

LFGraph's improvement increases with cluster size. This indicates intelligent partitioning is prohibitive at all cluster sizes. In a small cluster, distributed graph processing is compute-heavy thus intelligent partitioning (e.g., in PowerGraph) has little effect. In a large cluster, intelligent partitioning can speed up iterations – however, the partitioning cost itself is directly proportional to cluster size and contributes sizably to runtime.

Figure 4.2: PageRank runtime improvement for Twitter graph (10 iterations), including partition time.

4.2.2 Memory Footprint

While PowerGraph stores in- and out-links and other book-keeping information, LFGraph relies only on in-links and publish-lists (Section 2.3). We used the `smem` tool to obtain LFGraph's memory footprint. For PowerGraph we used the heap space reported in the debug logs. Figure 4.3 shows that LFGraph uses 8x to 12x less memory than PowerGraph.

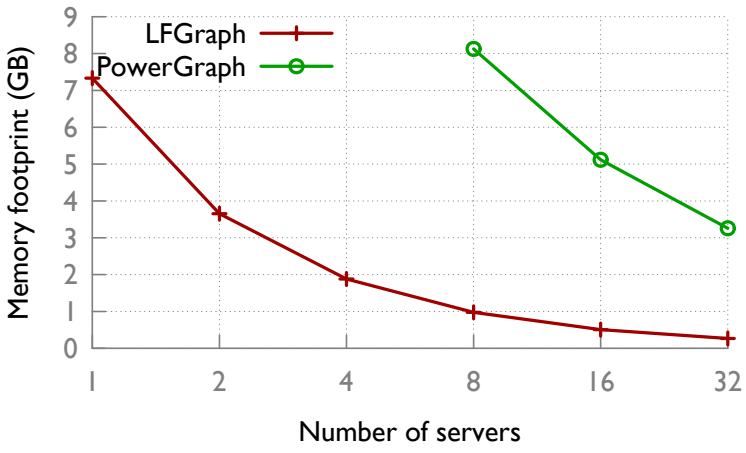


Figure 4.3: Memory footprint of LFGraph and PowerGraph

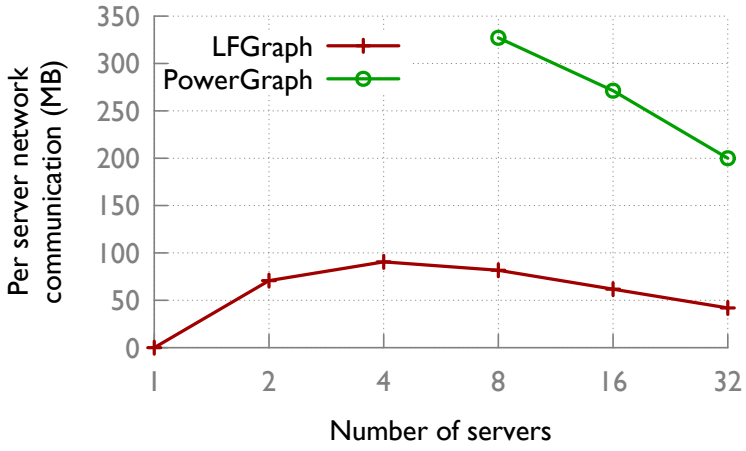


Figure 4.4: Network communication for LFGraph and PowerGraph

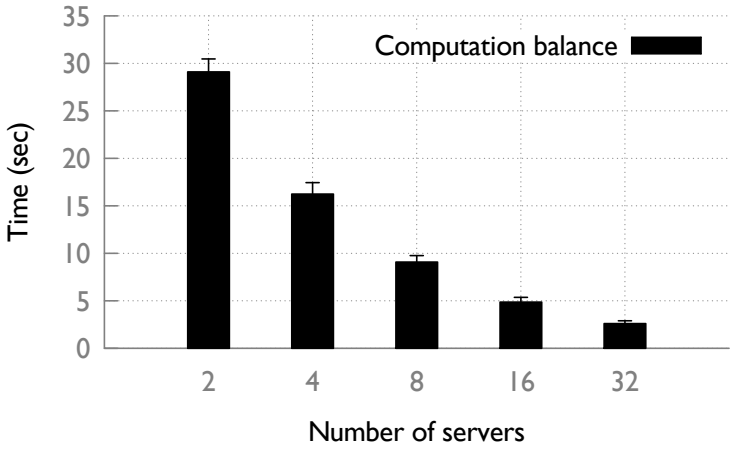
4.2.3 Communication Overhead

Figure 4.4 shows that LFGraph transfers about 4x less data per server than PowerGraph – this is consistent with our analysis in Section 2.4. We also notice that the LFGraph's communication reaches a peak at about 4 servers. This is because the per-server overhead equals the total communication overhead divided by the number of servers. As the cluster size is increased, there is first a quick rise in the total communication overhead (see Section 2.4 and Figure 2.3). Thus the per-server overhead rises at first. However as the total communication overhead plateaus out, the cluster size increase takes over, thus dropping the per-server overhead. This creates the peak in between.

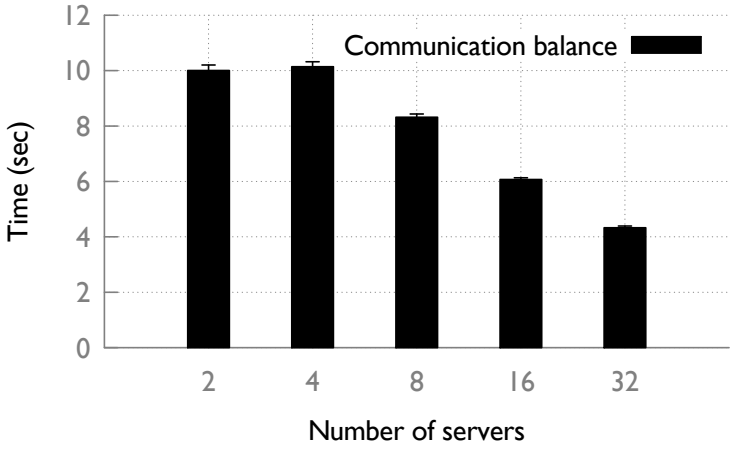
Finally, we observe that although the total communication overhead rises with cluster size (Figure 2.3), in the real deployment the per-iteration time in fact falls (Figure 4.1). This is because of two factors: i) communication workers schedule network transfers in parallel, and ii) Emulab offers full bisection bandwidth offered between every server pair. Since (ii) is becoming a norm, our conclusions generalize to many datacenters.

4.2.4 Computation and Communication Balance

As a counterpart to Section 2.5, Figure 4.5 shows, for different cluster sizes, the average overhead at a server (measured in time units) along with the standard deviation bars. The small bars indicate good load balance in LFGraph. The communication bars are smaller than the computation bars primarily due to the stability of Emulab's VLAN.



(a) Computation Balance



(b) Communication Balance

Figure 4.5: Computation and communication balance in LFGraph

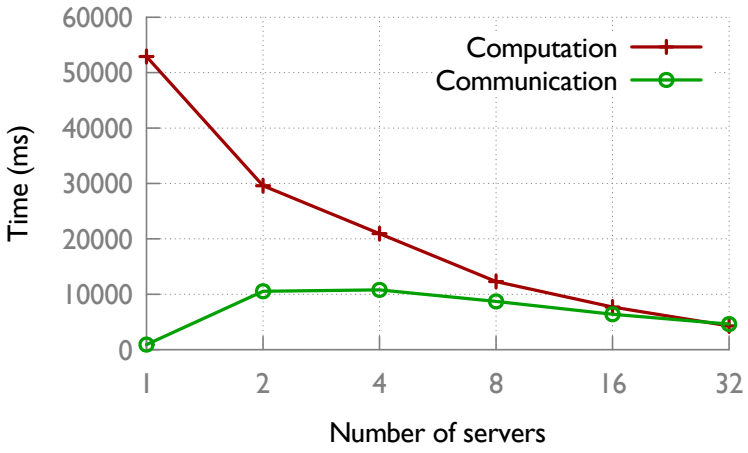


Figure 4.6: Communication and computation split of PageRank computation

4.2.5 Computation vs. Communication

Figure 4.6 shows the split between computation and communication at different cluster sizes. First, computation time decreases with increasing number of servers, as expected from the increasing compute power. Second, communication time variation with cluster size mirrors the per-server network overhead plotted in Figure 4.4 and discussed earlier. Third, compute dominates communication in small clusters. However, at 16 servers and beyond, the two phases take about the same time as each other. This indicates that the importance of balancing computation and communication load in order to achieve the best runtime. LFGraph achieves this balance.

4.2.6 Improvement Breakdown

In order to better understand the sources of improvement in LFGraph we plot the overall runtime of 10 iterations of PageRank along with the time spent in the communication phase for both LFGraph and PowerGraph in Figure 4.7. Note that although communication and computation are disjoint

in LFGraph, they overlap in PowerGraph. So, it is impossible to accurately measure the communication-computation split in PowerGraph. Therefore, we assume that the time spent in communication is proportional to the amount of data transferred and calculate the communication time of PowerGraph. The plot indicates that LFGraph's performance improvement is due to shorter communication and computation phases. First, LFGraph transfers less data. So, communication phase is shorter. Second, LFGraph processes only incoming edges. So, its computation phase is shorter compared to that of PowerGraph.

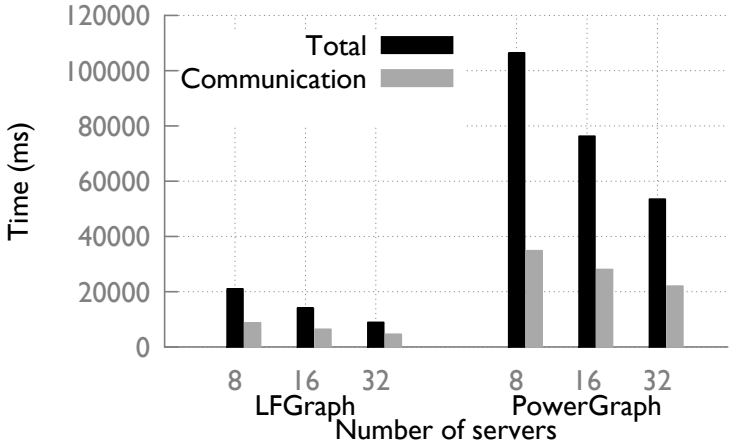


Figure 4.7: Breakdown of performance gain in LFGraph compared to PowerGraph

One important observation from the plot is: although we expected the computation phase to be 2x faster in LFGraph compared to PowerGraph, the improvement is ranging from 5x to 7x. This is because communication and computation are overlapping in PowerGraph. So, the increased communication overhead is negatively affecting the computation time as well. In addition, because communication and computation are disjoint in LFGraph, we can optimize communication by batching data transfers. On the other hand, PowerGraph is unable to perform such optimizations. So, the time re-

quired for communication is much higher in PowerGraph than what is shown in the plot.

4.2.7 PageRank on Undirected Graph

We repeat the PageRank benchmark on the Twitter Graph by making the edges undirected. This is important for two reasons. First, we showed in Section 2.4.1 that LFGraph's improvement over PowerGraph is the largest when the incoming and outgoing edge lists of vertices are disjoint. The improvement is the lowest when the incoming and outgoing edge lists of vertices overlap completely, i.e., the graph is undirected. So, this experiment provides more insight on LFGraph's runtime and communication overhead for undirected graphs. Second, it is important to compare the memory overhead of LFGraph and PowerGraph for undirected graphs because PowerGraph already stores both incoming and outgoing edges. So, in case of an undirected graph PowerGraph's memory overhead for storing the edge list should be equal to that of LFGraph's.

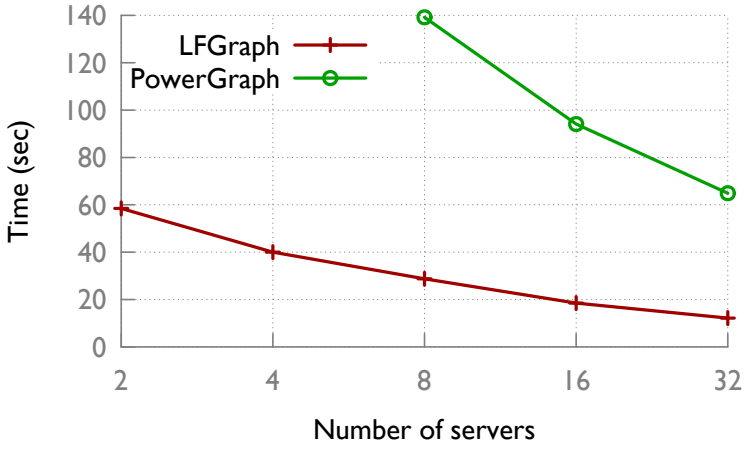


Figure 4.8: PageRank runtime comparison for undirected Twitter graph (10 iterations), ignoring partition time.

Figure 4.8 shows PageRank runtime on the undirected version of the Twit-

ter graph for 10 iterations, ignoring partition time. As before, PowerGraph was unable to load the graph on less than 8 servers. Interestingly, LFGraph was unable to run the benchmark on a single server, because of the increased size of the undirected version of the graph. We see that even for undirected graphs LFGraph runs faster compared to PowerGraph. However, the improvement is 5x as opposed to the 6x observed in the directed graph case.

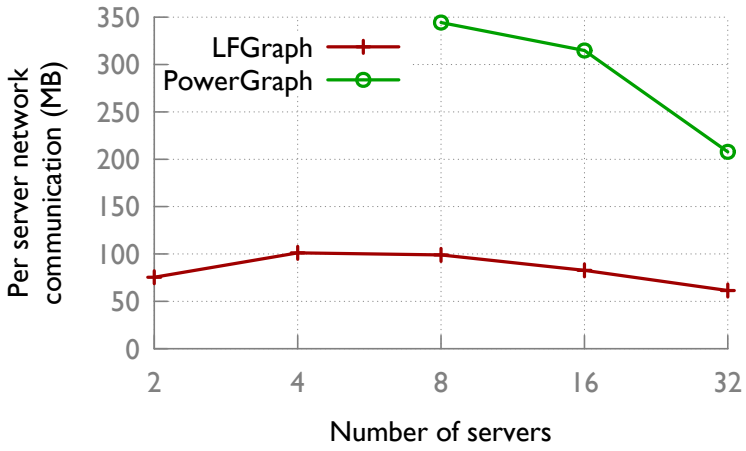


Figure 4.9: Network communication for LFGraph and PowerGraph on undirected Twitter graph

The reduction in improvement is justified by Figure 4.9, where we plot the communication overheads of LFGraph and PowerGraph. For undirected graphs, LFGraph's communication overhead is up to 3.4x lower compared to PowerGraph's. Note that for directed graphs LFGraph exhibited 4.8x lower communication overhead compared to PowerGraph. This behavior conforms to our analysis in Section 2.4.1.

Finally, in Figure 4.10, we show the memory overhead for LFGraph and PowerGraph for the undirected Twitter Graph. Contrary to our speculation, we observe that LFGraph's memory overhead is significantly lower compared to PowerGraph's memory overhead. This is for two reasons. First, PowerGraph has to maintain location information of the mirrors, which requires

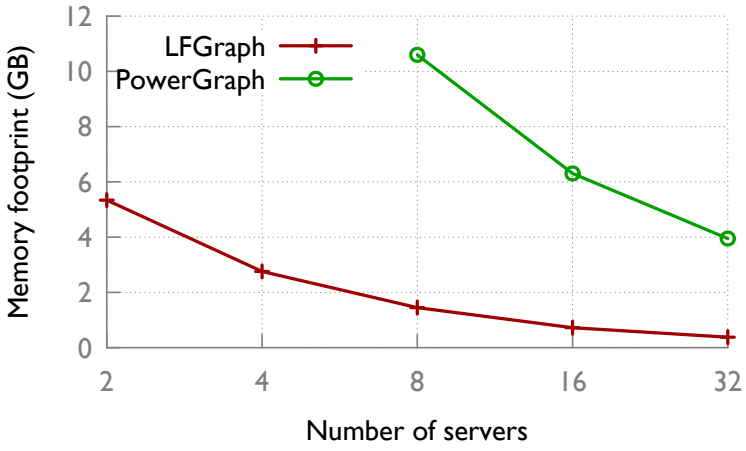


Figure 4.10: Memory footprint of LFGraph and PowerGraph for the undirected Twitter graph

additional memory. Second, PowerGraph maintains two lists even for undirected graphs when the incoming and outgoing lists are identical. Therefore, we conclude that LFGraph performs better than PowerGraph even for undirected graphs for the PageRank benchmark.

4.3 SSSP Benchmark

We ran the SSSP benchmark on the Twitter graph. The benchmark ran for 13 iterations. Figure 4.11 shows the comparison between LFGraph and the three PowerGraph variants, ignoring the partitioning time for PowerGraph's Oblivious and Batch strategies.

First, we observe that LFGraph successfully ran the benchmark on a small cluster (4 servers and less), while PowerGraph could not, due to its memory overhead. Second, unlike in PageRank (Section 4.2), LFGraph and PowerGraph are comparable. At 8 servers, LFGraph's performance is similar to that of PowerGraph's random placement but worse than PowerGraph's intelligent placement strategies. This is due to two factors: i) SSSP in-

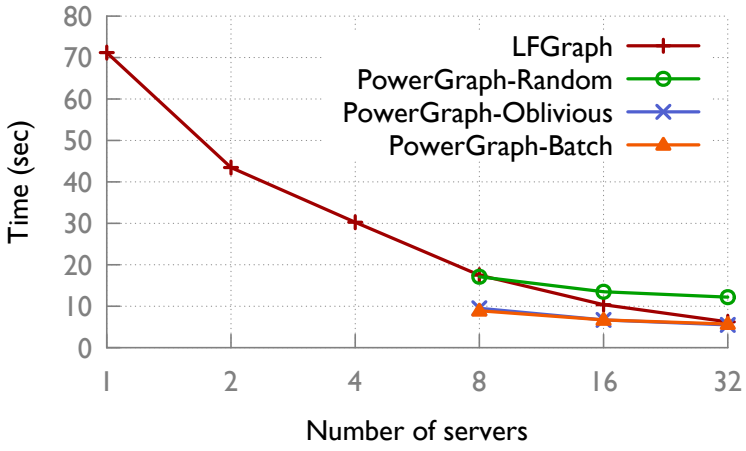


Figure 4.11: SSSP runtime comparison for Twitter graph, ignoring partition time

curs less communication than PageRank, especially in later iterations, and ii) LFGraph does not store out-links, thus unlike PowerGraph it cannot activate/deactivate vertices for the next iteration. Recall that in LFGraph, a vertex has to iterate through all of its in-neighbors to check which were updated in the previous iteration.

At 16 servers and beyond, LFGraph is better than PowerGraph's random placement. At 32 servers LFGraph exhibits similar runtime as the Oblivious and Batch strategies. This is because communication starts to dominate computation at these cluster sizes.

Finally, Figure 4.12 shows LFGraph's improvement over the PowerGraph variants, when including the partitioning time. We observe up to a 560x improvement.

We conclude that for SSSP-like analytics LFGraph is almost always preferable to PowerGraph, with the only exception being the corner-case scenario where the graph is loaded once and processed multiple times and in a cluster that is medium-sized (8–16 servers in Figure 4.11).

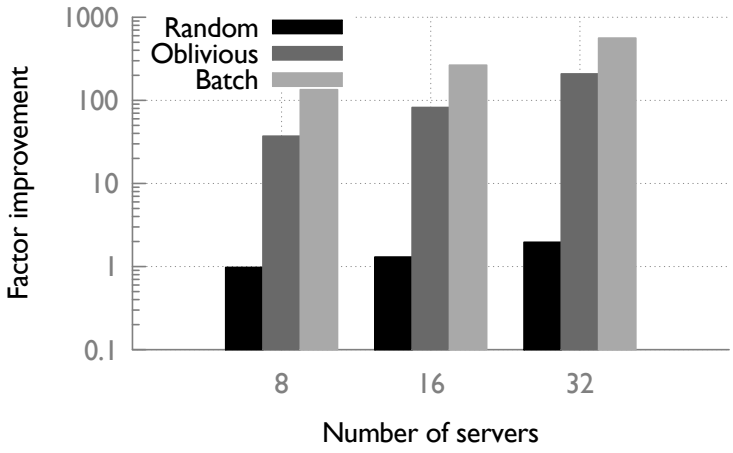


Figure 4.12: SSSP runtime improvement for Twitter graph, including partition time

4.4 Larger Graphs

We create 10 synthetic graphs varying in number of vertices from 100M to 1B, and with up to 128B edges. We run the SSSP benchmark on it. These graphs are generated by choosing out-degrees from a log-normal distribution with $\mu = 4$ and $\sigma = 1.3$, with out-neighbors selected at random. To avoid the network overhead for graph creation, we cap the in-degree of each vertex at 128 and choose in-neighbors at random such that the probability of choosing a vertex as an in-neighbor follows the aforesaid log-normal distribution. Other papers [27] have used similar graphs for evaluating their systems.

We performed this experiment on a slightly different setup – a 12 server Emulab cluster with each server containing four 2.2 GHz E5-4620 Sandy Bridge processors, 128 GB RAM, and connected via a 10 GigE network.

Figure 4.13 shows the results for LFGraph. We juxtapose the Pregel performance numbers from [27] – those Pregel experiments used 300 servers with 800 workers. In comparison, our LFGraph deployment with only 12 servers with 96 workers uses around 10x less compute power. Even so we observe an overall improvement in runtime. The cores per server ratio is higher in the

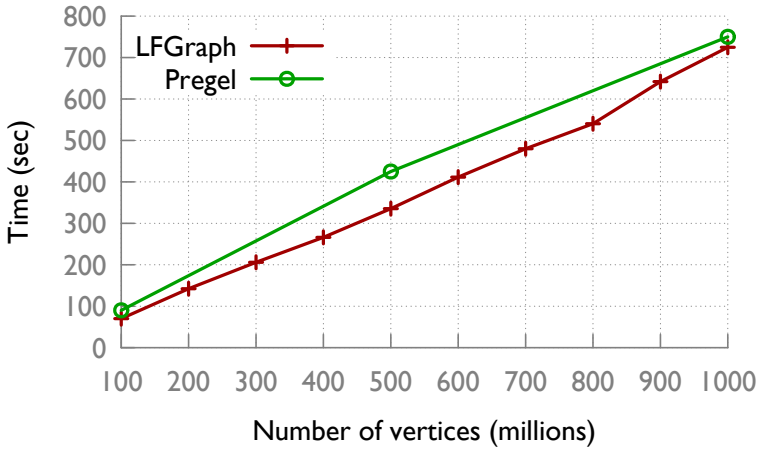


Figure 4.13: SSSP runtime for a synthetic graph

LFGraph setting – we believe this is in keeping with current architecture and pricing trends. Thus we conclude that LFGraph can perform comparably to industrial-scale systems while using only a fraction of the resources.

4.5 Undirected Triangle Count Benchmark

Finally, we present results from undirected triangle count benchmark on the undirected version of the Twitter graph in Figure 4.14. Here, the values associated with the vertices are the edge-lists. So, the sizes of values are variable and large compared to PageRank (a single floating point) and SSSP (a single integer). So, this benchmark is communication intensive.

Due to the extensive resource requirement of Triangle Counting computation, we performed this experiment on a beefier cluster – an 8 server Emulab cluster with each server containing four 2.2 GHz E5-4620 Sandy Bridge processor, 128 GB RAM, and connected via a 10 GigE network. We make two important observations here. First, LFGraph outperforms PowerGraph in terms of runtime by almost a factor of 2. Second, the computation could not be performed on a single machine in case of PowerGraph due to its extensive

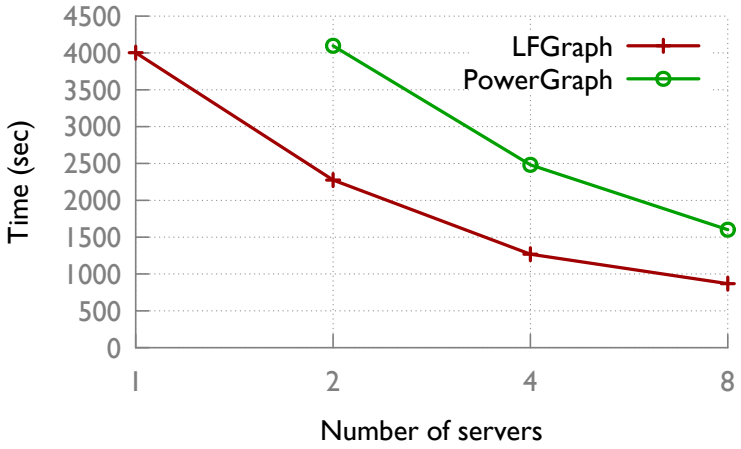


Figure 4.14: Triangle Counting on the undirected Twitter Graph

memory requirement. Thus, LFGraph outperforms PowerGraph in terms of both runtime and memory overhead for communication heavy workloads.

4.6 Summary

We have presented LFGraph, a system for fast, scalable, distributed, in-memory graph analytics. We have shown analytically that LFGraph's power-law-aware techniques incur lower communication overhead than existing systems, and exhibit good load balance. These techniques help LFGraph to offer low pre-processing overhead, low memory footprint, and good scalability. LFGraph is faster than the best existing system by 380x for PageRank and by 560x for SSSP. We have also shown, analytically and experimentally, that intelligent graph partitioning schemes yield little benefit and are prohibitive.

5 EXPERIMENTAL EVALUATION OF BONDHU

In this chapter, we present the experimental evaluation of the Bondhu system. First, we present a visualization of disk block access patterns to show that the graph-aware placement techniques in Bondhu cluster disk accesses. Later, we show the effectiveness of clustered disk accesses in lowering the response time for OSN queries. We also present results by varying the granularity of clusters. Our experimental results indicate that while taking the small world nature (i.e., community structure) into account yields clear benefits, models with more complexity beyond the graph structure may yield low additional benefit.

5.1 Experimental Setup

We use the Facebook New Orleans dataset collected in [78]. This dataset contains 63731 users and 817090 links. We assign appropriate weights to the social graph according to our uniform, preferential, and overlap models. Unless otherwise specified the experiments are based on the uniform model.

We run two instances of Neo4j that store the above OSN – one with the integrated Bondhu system handling the data layout and the other one is the unmodified Neo4j. We call the data layout scheme of the unaltered Neo4j the $default$ layout. Based on the method used in the community detection module, the Bondhu system has two data layout schemes built-in: ParCom and ModCom (see Section 3.3 for details).

We use two metrics to evaluate the data layout schemes of the Bondhu system. The first metric is the cost as defined in Section 3.2 (Equation 3.1). The cost metric measures the spatial clustering of the friends of a user on the disk. Therefore, a lower cost means that the data of related users are placed close by on the disk. Thus, operations like listing friends and wall posts will be faster. Our second metric is response time. This measures the time to fetch data blocks from all the friends of a random user. This captures the performance of an application with our layout schemes. An improvement in the response time metric suggests that the disk is able to handle more requests per unit time and that the user-perceived delay in getting the response to a request is reduced. We next describe the workload we use to measure the response time metric.

Our workload captures the event of listing the friends of a user, which is a very common operation in an OSN. To do this we build a sample OSN application on top of Neo4j. First, we populate the Neo4j database with the social graph. Next, we store a data block (*property* in Neo4j) for each user in the graph. The Bondhu system handles the data layout automatically beyond that point. Next, we write an automated script that logs into the system as a random user and fetches the data blocks for all the friends of that user. We measure the response time for the whole operation. To make sure that the response time is not adversely affected by other processes accessing the disk at the same time, we carry out the same operation 6 times and take the minimum. We repeat this for 1500 random users. We use the same workload for all of our experiments except for the one on the effect of different models (Section 5.8).

To ensure that the data is served from the disk (and not from the previous cached results in the memory) we flush the memory as follows: First, we use

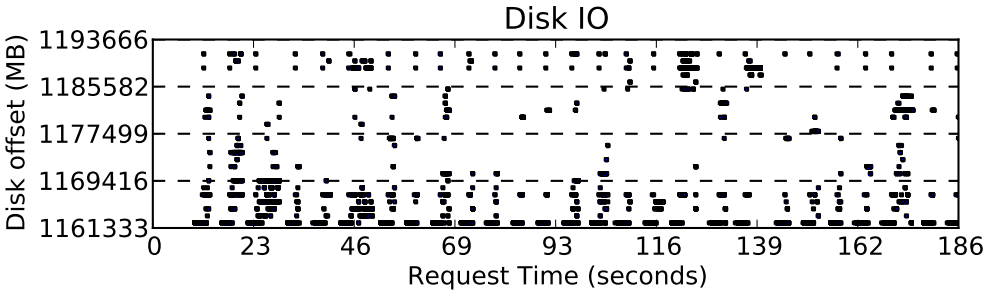


Figure 5.1: Blocks accessed in Neo4j with the Bondhu system handling data layout. Compare with Figure 3.1 (default approach).

the *sync* command to write any buffered data to disk. Then, we use the *drop_caches* mechanism in the Linux kernel to drop the pagecache, dentries, and inodes from the memory, causing the memory to be free from any cached data. All our experiments are conducted on an HP DL160 compute block with 2 quad core Intel Xeon 2.66 GHz processors, 16 GB of memory, and 2 TB of storage.

5.2 Visualization of Block Access Patterns

To contrast with the disk block access patterns of the default layout presented in Section 3.1, we repeat the same experiment with Bondhu enabled Neo4j system. Here we use ParCom with 64 communities. Per user data size is 400KB as before. We conduct our workload based measurements and trace the block level I/O for each user request. The visualization of the trace is presented in Figure 5.1. Each dot shows a read request, its disk offset, and time of request. Here, we observe a significantly better disk block access pattern compared to Figure 3.1. In Figure 3.1 the block accesses were scattered, whereas in Figure 5.1 the block accesses are clustered (prominent at 23, 46, 116–139 seconds). This suggests that the Bondhu system is clustering the

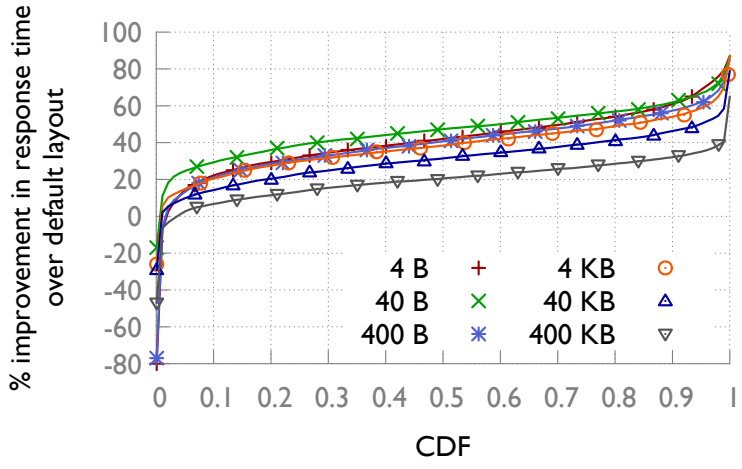
related friends' data close by on the disk. This translates to less disk arm movement and thus faster seek and response time.

5.3 Effect of Data Size

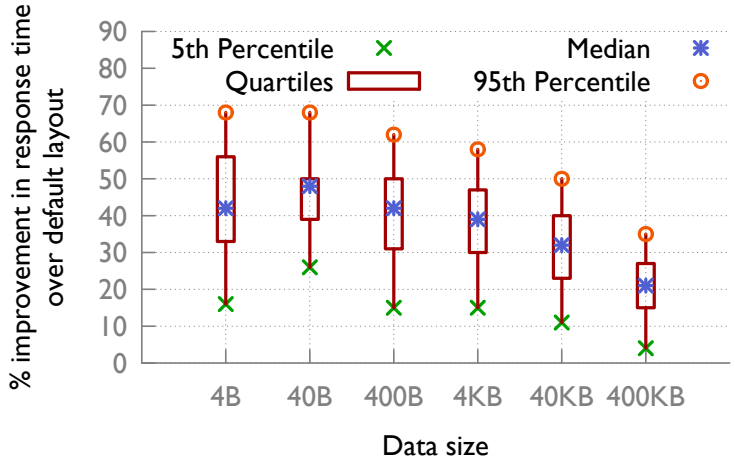
In an OSN application the data associated with a particular user can be of different sizes, e.g., it may contain any of name, address, profile picture, wall posts, etc. Therefore, it is important to see the effect of varying user data block sizes on the performance of the layout algorithms. First, we examine the effect of varying data block sizes on the response time metric. Then we present a scatter plot to show the correlation between the improvement in the cost metric and the improvement in the response time metric. This shows to what extent the improvement in data locality translates to the improvement in response time.

For this experiment we create data blocks of 4B, 40B, 400B, 4KB, 40KB, and 400KB for each of the users and conduct our workload based measurement. We use ParCom with 64 communities, compare it with the default layout, and plot the improvement in the response time metric (the lower the response time compared to the default layout, the more is the improvement). We plot the CDF of the improvement for the different data sizes in Figure 5.2(a) and the 5th percentile, quartiles, and the 95th percentile of the same results in Figure 5.2(b).

We see a 22% to 48% median improvement in response time compared to the default layout across various data sizes. The Bondhu layout manager performs best when the user data size is 40B. When the data sizes increase from 4B to 40B we see an increase from 42% to 48% in the median improvement compared to the default layout. Beyond that, the improvement percentage



(a) CDF



(b) 5th percentile, quartiles, and 95th percentile

Figure 5.2: Percentage of improvement in response time compared to the default layout for various data sizes (without caching)

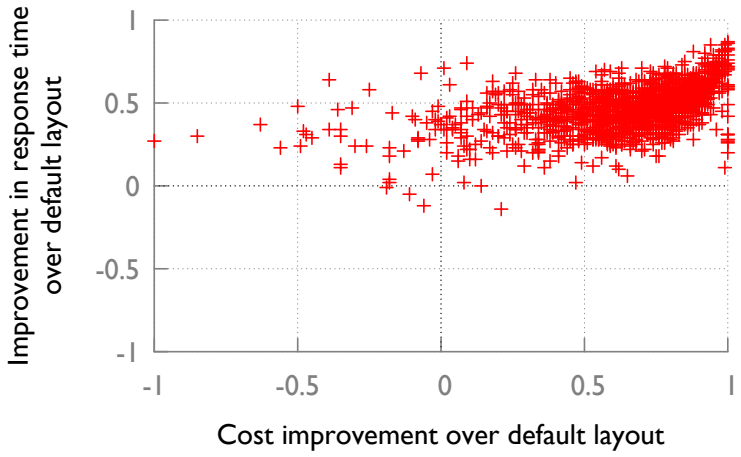
decreases and at 400KB the median improvement reaches 22%.

The reasoning for the above behavior is as follows. The native file system reads data in chunks of 4KB blocks. Therefore, when user block sizes are small, a file system read fetches multiple users' data together. For example, with 4B user block size, a read yields around 1024 users' data. With 40B user block size, a read yields around 102 users' data, and with 400B user block size, a read yields around 10 users' data. Due to the randomness of the data placement scheme of the default layout, the expected number of friends

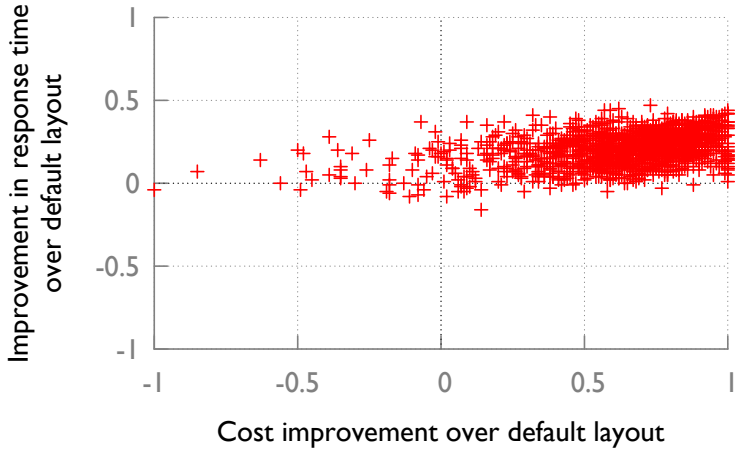
present per read decreases by a factor of 10 when data block sizes grow from 4B to 40B to 400B. For Bondhu, however, the expected number of friends present per read does not decrease much when data block sizes grow from 4B to 40B. This is due to the clustering property of Bondhu. In contrast, when the block sizes grow from 40B to 400B, the expected number of friends present in per read decreases dramatically. Therefore, we see the drop in improvement after 40B.

In summary, when user data size is smaller than the file system block size, the improvement is high due to fact that a single file system read yields more correlated data. So, the number of seeks required to fetch all friends' data is reduced. This phenomenon begins to vanish when user data sizes grow larger than the file system block size. Beyond that point, the Bondhu system improves performance by reducing the seek distance.

Next, Figure 5.3 examines the correlation between the improvement in the cost metric and the improvement in the response time. This shows how the smart placement decision of the Bondhu system translates to better application-level performance. As defined in Section 3.2, the cost metric for a user is the sum of differences between the user and her friends' data location. We calculate the cost metric for the users using the placement in both the default layout and the ParCom layout. A larger cost denotes that the friends of a user are far away in the disk. We then calculate the fraction of improvement by using the ParCom layout scheme of the Bondhu system over the default layout. For the corresponding users we measure the fraction of improvement in response time metric and plot the results using a scatter plot. The results are presented in Figures 5.3(a) and 5.3(b) for two different data sizes. Figure 5.3(a) shows good correlation since most points are along the $x = y$ line. For Figure 5.3(b) the correlation is less prominent because of



(a) Data size: 40B



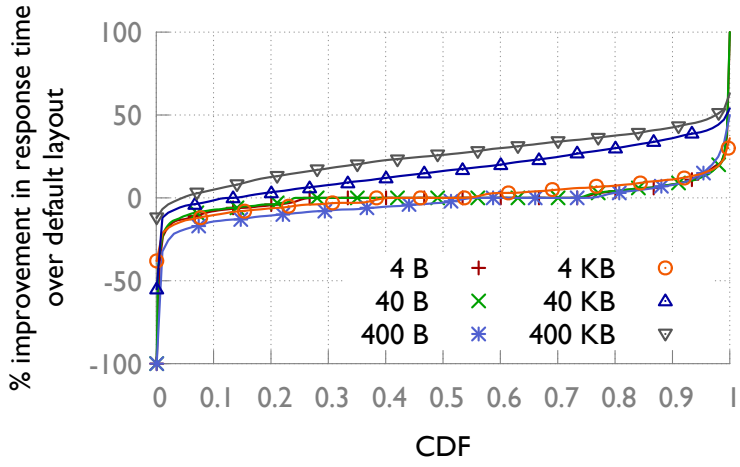
(b) Data size: 400KB

Figure 5.3: Correlation between cost improvement and response time improvement (without caching)

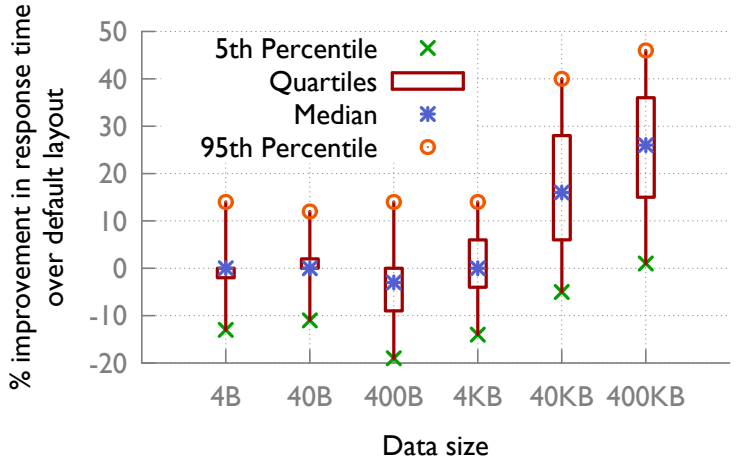
the prior discussion.

5.4 Effect of Caching

In the previous section we ensure that all the requests are served from the disk and not from the memory. However, serving results from the memory reduces the response time by a large fraction for any application. So, we enable caching for both Neo4j and Bondhu. The results presented in this



(a) CDF



(b) 5th percentile, quartiles, and 95th percentile

Figure 5.4: Percentage of improvement in response time compared to the default layout for various data sizes (with caching)

section examine the effect of caching on the response time metric.

We use the same workload as discussed earlier, but without flushing the cache between successive user requests. A user issues 10 successive requests to fetch the data blocks of all of her friends. As before we conduct this experiment for 1500 randomly selected users.

As with the previous experiment, we plot the CDF of the improvement in response time for the different data block sizes in Figure 5.4(a) and the 5th percentile, quartiles, and the 95th percentile of the same results in Fig-

Figure 5.5: Correlation between cost improvement and response time improvement (with caching)

ure 5.4(b). When the data size is small we do not see much improvement using our layout scheme. As the data sizes increase from 4KB to 40KB to 400KB the benefit of using the Bondhu system kicks in as seen by the rise in median response time improvement from 0% to 16% to 26% respectively. This is because when the data sizes are small, the information of all the users can be kept in memory. Therefore, requests for data can be readily served from the memory for the default layout as well as for the Bondhu layout schemes. When the data size grows beyond some threshold (40KB here), all the user data blocks cannot be kept in memory. If the data cannot be served from memory, it has to be fetched from disk and thus the previous section's described behavior kicks in.

To investigate whether the improvement in response time for larger data sizes is indeed due to the placement decisions by the Bondhu system, we present a scatter plot of the improvement in response time vs. the improvement in the cost metric in Figure 5.5. This is similar to the one presented in the previous section but with caching enabled. We observe a fair amount of correlation between the improvement in the two metrics in this case as

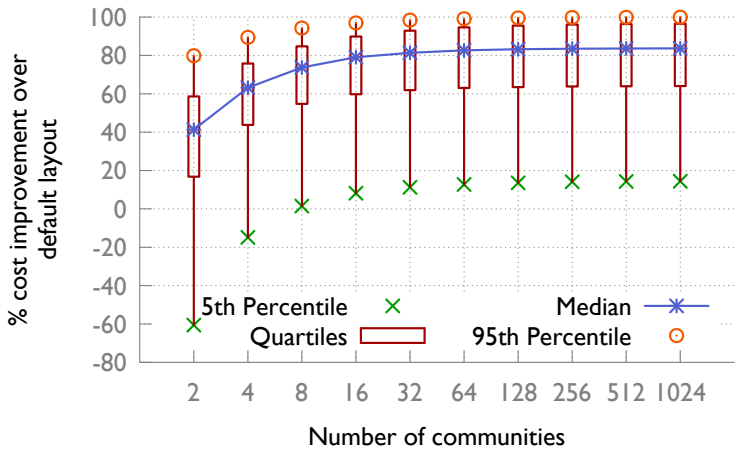
well. However, the correlation is not as strong as in Figure 5.3(b). With caching enabled, a fraction of the friends' data will be readily available in the memory. For the already cached data no disk read will be performed.

In summary, the worst case median improvement achieved by Bondhu is 0% (small data sizes with caching) and the best case improvement in 48% (medium data sizes without caching). Thus, it is always preferable to use Bondhu.

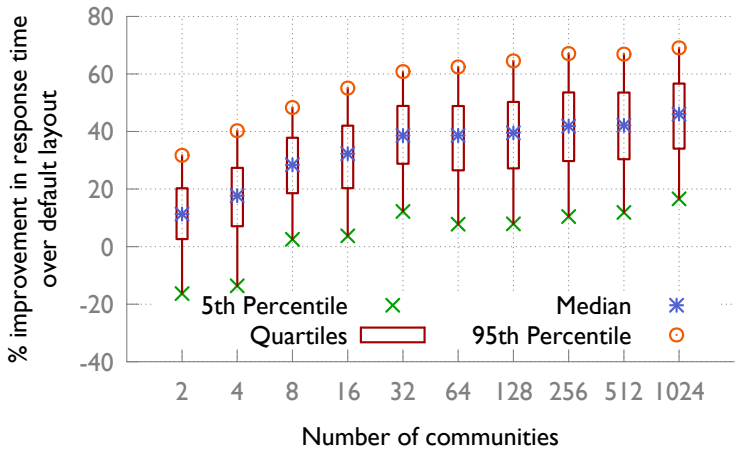
5.5 Effect of Number of Communities in ParCom

One of the parameters that can be tuned in ParCom is the number of communities. The fewer the number of communities, the larger the size of a community. For instance, with 1 community, the layout decision is solely handled by the intra-community layout module. With an increase in the number of communities, the inter-community layout module influences layout more. For a given social network graph, we desire to tune the number of communities in such a way that the best disk layout is obtained.

We vary the number of communities in ParCom and examine the improvements in the cost metric and in the response time metric over the default layout. The workload is the same as discussed earlier and the data size per user is 4KB. The results are presented in Figures 5.6(a) and 5.6(b) respectively. In Figure 5.6(a) we see that as the number of communities increases from 2 to 32 we experience a steady improvement in the cost metric. When we have fewer communities, the intra-community detection module is mostly responsible for the layout and the Bondhu system does not capture the community structure of the social graph. Therefore, the improvement grows quickly as the number of communities is increased. However, this curve



(a) Percentage of cost improvement over default layout



(b) Percentage of improvement in response time over default layout

Figure 5.6: Performance of ParCom

hits a knee at 64 communities and plateaus out thereafter. This is because the community detection module has lower marginal utility in finding more community structure in the graph towards the right end of the plot.

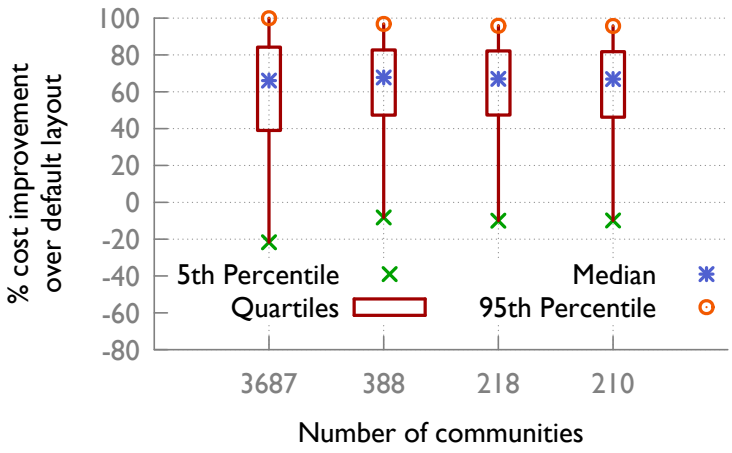
A similar pattern is observed in Figure 5.6(b), where we plot the improvement in the response time metric over the default layout for different number of communities. When the number of communities is 2, the median improvement in the response time metric is around 11% for ParCom and this grows quickly. The knee is reached at 32 communities, where the response time of

ParCom is 40% lower than that of the default layout. The reasoning is the same as in the previous paragraph.

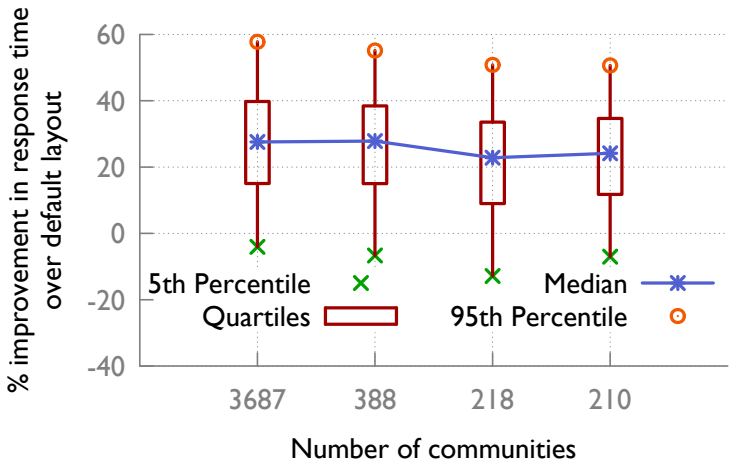
5.6 Performance of ModCom

We now focus on ModCom and examine the improvements in the cost metric and the response time metric over the default layout. Unlike ParCom we cannot set the number of communities in ModCom since the number of communities evolve depending on the structure of the social graph. However, ModCom produces communities at different granularities. Recall that the algorithm is iterative – at level 0, there are as many communities as the number of nodes. Level $(i + 1)$ combines the communities of level i, and produces fewer communities. We configure the Bondhu system so that it can organize the disk layout based on the communities found at any level. For example, if we set level=2, then the community detection module produces 388 communities which is then fed to the intra- and intra-community layout modules in succession. The workload and the metrics considered are same as the other experiments. Data block size for each user is set to 4 KB.

In Figure 5.7(a) we present the improvement in cost metric compared to the default layout for varying number of communities found by the community detection module. We observe that unlike ParCom, the median improvement ($\approx 67\%$) does not change much by varying the number of communities. This is because ModCom does not produce a community until it has found a good one (based on the value of the modularity). For the same reason, a flat pattern is observed for the response time metric in Figure 5.7(b).



(a) Percentage of cost improvement over default layout



(b) Percentage of improvement in response time over default layout

Figure 5.7: Performance of ModCom

5.7 Organ Pipe Layout

Next, we compare our layout algorithm with the popular organ pipe algorithm [73, 74], which is used in multimedia file systems. Given a set of records $R_1, R_2, \ldots, R_N$ with global access probabilities $P_1, P_2, \ldots, P_N$, and $\sum_{i=1}^{N} P_i = 1$, the organ pipe algorithm places the record R_i with the largest P_i in the middle and then iteratively places the record with the next largest P_i alternatively to the left and to the right of the already placed record(s). So, according to the organ pipe scheme, the most popular user (the user with

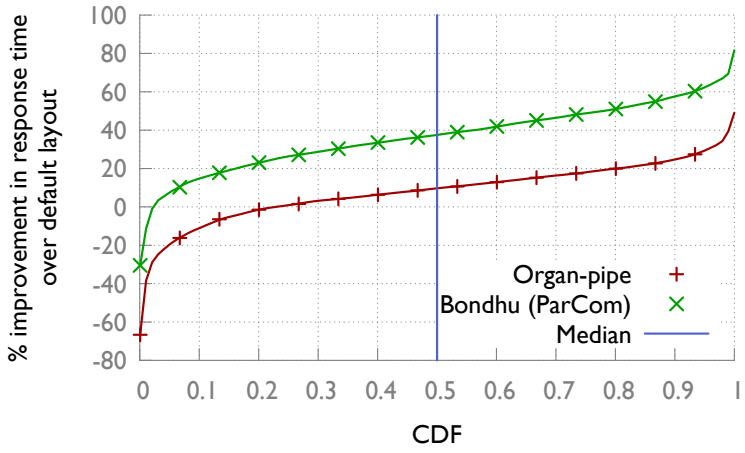


Figure 5.8: Comparison with organ pipe layout

the largest edge degree) is placed in the middle and the users with the next largest edge degrees are placed alternately to the left and to the right of the already placed user(s).

We modify the Bondhu system to organize data according to the organ pipe scheme and compare it with ParCom (number of communities=64). Figure 5.8 plots the CDF of the improvements of the response time metric compared to the default layout for both. The data block size for each user is $4KB$ and the workload is the same as the preceding experiments.

The organ pipe is better than the default layout by 10% (on average), but ParCom outperforms the default layout by 38%. The organ pipe scheme assumes that popular users are popular across the system, which is not valid for an OSN. An OSN has a very specific community structure and in this structure popular users are popular only among their friends.

5.8 Effect of Different Models

So far all the experimental results are based on the uniform model. In this section we present results using the different models presented in Section 3.5:

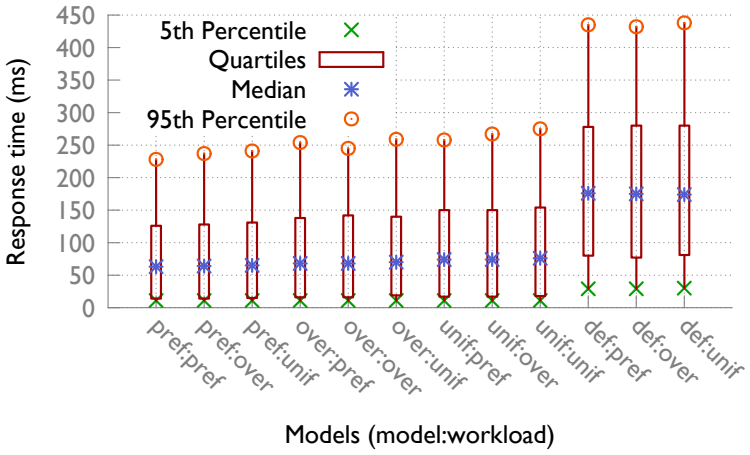


Figure 5.9: Effect of different models

i) the preferential model, ii) the overlap model, and iii) the uniform model. To provide as a baseline for comparison we also present results using the default layout. We use the same social graph as the previous experiments. Data block size for each user is 4KB. The Bondhu system takes the model as the input, creates a layout using that model, and organizes the data according to that layout.

We use 3 different workloads based on the graph models. First, in the uniform workload we randomly select a user who issues a request to access one of her friends' blocks at random. Second, in the preferential workload the randomly selected user issues a request to access a friends' data blocks with probability proportional to the friend's degree. Third, in the overlap workload the randomly selected user issues a request to access one of her friends' data blocks with probability proportional to the number of common friends with the friend. In each of these workloads a user issues 1000 successive requests and the response time is measured. Each measurement is taken 10 times and we take the minimum response time. We conduct this experiment for 1000 users in total.

We run each of the 3 workloads on the 4 different layouts and present the

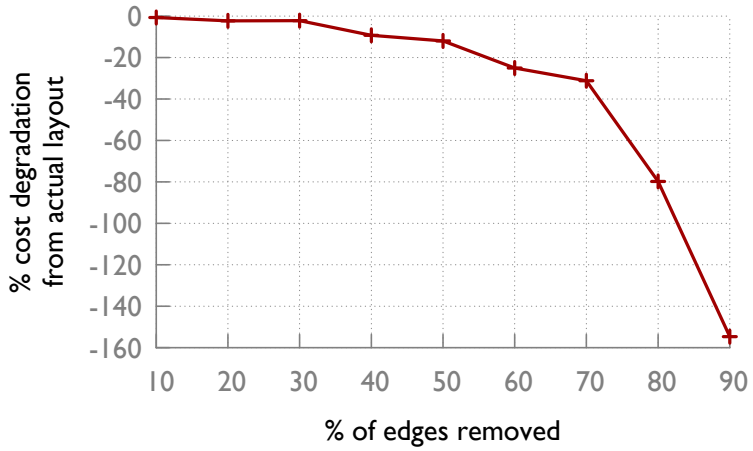
results in Figure 5.9. Each run of the experiment is denoted by (*model* : *workload*), where model denotes the models we use: {preferential, overlap, uniform, default} and workload denotes the workloads we use: {preferential, overlap, uniform}. We plot the 5th percentile, quartiles, and the 95th percentile of the response time for all of the runs.

We make three observations from this plot. First, the default layout performs twice worse than any of the other models (median response time: 175ms). Second, the performance of the layout produced by the uniform model is quite comparable to the performance of the layout produced by the preferential and overlap models. Third, the performance of a specific layout does not vary much over the different workloads.

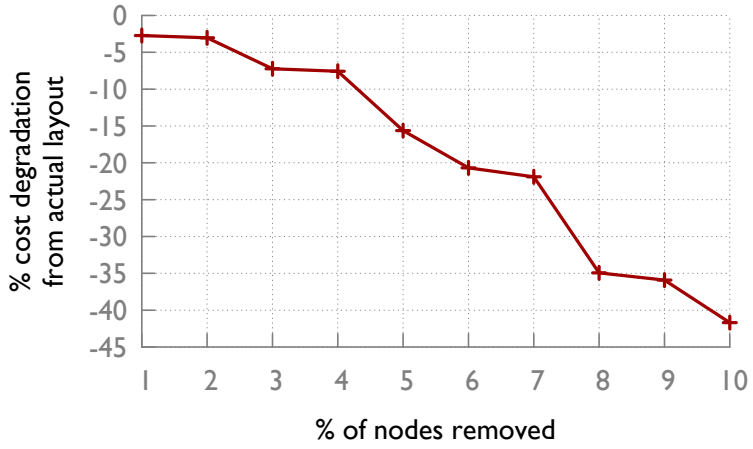
One directional conclusion from these observations is that although it is possible to create complex models (e.g., [106]) to capture user interactions in a social network, often the simplest model (such as the uniform model) is sufficient to make important disk layout decisions. Taking more complex models into account may yield little added benefit for the amount of effort involved. The social graph structure is the biggest influence on disk performance.

5.9 Effect of OSN Evolution

With the evolution of the OSN, the layout may need to be reorganized to reflect the new social graph. While techniques for modifying disk layout incrementally are beyond the scope of this thesis, we examine how frequently the layout needs to be reorganized. To do this, we first create older versions of the graph by removing edges and nodes at random respectively. Next we use the layout obtained from that older version to host the latest version of the graph. The nodes in the latest graph that are not present in the older



(a) Edge removal



(b) Node removal

Figure 5.10: Effect of OSN evolution: older layout performance

graph are placed sequentially after the old layout to produce the new layout. We measure the percentage of degradation in the cost metric from using the older layout instead of the layout based on the latest graph. Intuitively, as long as the percentage of degradation is reasonable, the old layout can be used and reorganization is not needed. The results are plotted in Figures 5.10(a) and 5.10(b) respectively.

Figure 5.10(a) shows that even when the layout is based on a graph with 40% edges removed from the current graph, the cost metric degrades by less

than 10%. Most social networks reach a plateau in terms of the number of nodes after a point [107]. Therefore, reorganization will be infrequent after that point. On the other hand, Figure 5.10(b) shows when the layout is based on a graph with only 5% nodes removed from the current graph, the degradation is greater than 15%. However, note that the performance of an old layout can only be as bad as the default layout. Therefore, although a layout based on 10% fewer nodes has 42% cost degradation compared to the layout based on the current graph, it still has around 72% cost improvement over the default layout (since the layout based on the actual graph has 80% cost improvement over the default layout, see Figure 5.6(a)). As seen in [107], the growth rate of Facebook was around 50% over a period of 9 months (Mar '09 - Dec '09). This suggests that even at this growth rate reorganization can be done infrequently while still doing better than the default layout.

5.10 Summary

We have presented power-law-aware techniques for disk layout organization for graph databases running OSN applications. We incorporated our techniques into the Neo4j graph database by building a layout manager called the Bondhu system. Experimental results with realistic workloads exhibited that our power-law-aware techniques achieve up to 48% improvement in cost and response time compared to the default layout. Our results also indicate that models with more complexity beyond the power-law graph structure may yield low additional benefit.

6 CONCLUSION AND FUTURE WORK

In this thesis, we have shown that techniques which leverage the structure of the power-law graph make graph computation faster and graph storage more efficient. In doing so we have presented the design and implementation of two systems. Our first contribution, LFGraph, is a distributed graph analytics framework. LFGraph's power-law aware techniques lower communication overhead and ensure communication and computation balance. These techniques make graph analytics faster. Our second contribution, Bondhu, is a disk layout manager for graph databases. Bondhu's power-law-aware placement techniques make disk accesses faster for social networking applications. These techniques lower response time for queries in graph databases.

Future Work: We suggest several directions for future research related to this thesis:

- LFGraph requires that sufficient memory is available in the cluster to store the graph and the associated values. However, this assumption might not hold with increasing graph sizes. In addition, users might not have access to a cluster with sufficient memory. So, new techniques should be explored which can perform *fast* graph analytics with *limited memory*. Disk-based analytics systems [38] fall short because they do not work in a distributed setting and cannot efficiently utilize the available memory in a cluster. In order to design new techniques for 'wimpy' clusters, careful consideration should be given to which part

of the graph and values should be stored in-memory and which part should be stored in disk for faster analytics.

- SSDs are becoming popular and cheaper as storage media [108]. Layout techniques which work for traditional hard disk drives are not suitable for SSDs, because SSDs are better at random accesses than hard disk drives. In addition, SSDs have limited write cycle. Recent works have focused on tuning key-value storage systems for SSDs [109]. So, further investigation is necessary to design layout techniques for graph databases on SSDs.

- In LFGraph, the graph computations are synchronous in nature and run to completion. However, in scenarios where a deadline is imposed to obtain a final result, users might be satisfied with partial or imprecise results. Techniques for computation and representation of partial results for graph computation are an interesting future direction.

- Other NoSQL storage systems such as key-value storage systems [110, 111] and in-memory storage systems [112, 113] are also used to store graph data [114, 115]. These systems might benefit by exploiting the structure of underlying data. Further investigation is necessary to effectively discover the structure of data in these unstructured storage systems and to utilize the discovered structure for better system performance.

REFERENCES

[1] R. Albert, "Scale-free networks in cell biology," *Journal of Cell Science*, vol. 118, pp. 4947–4957, 2005.

[2] H. Jeong, B. Tombor, R. Albert, Z. N. Oltvai, and A. L. Barabási, "The large-scale organization of metabolic networks," *Nature*, vol. 407, pp. 651–654, 2000.

[3] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 79, no. 8, pp. 2554–2558, 1982.

[4] S. R. Proulx, D. E. L. Promislow, and P. C. Phillips, "Network thinking in ecology and evolution," *Trends in Ecology and Evolution*, vol. 20, no. 6, pp. 345–353, 2005.

[5] "Facebook," http://www.facebook.com.

[6] "Twitter," http://twitter.com.

[7] "LinkedIn," http://www.linkedin.com/.

[8] T. Lux and M. Marchesi, "Scaling and criticality in a stochastic multi-agent model of a financial market," *Nature*, vol. 397, pp. 498–500, 1998.

[9] B. G. Ryder, "Constructing the Call Graph of a Program," *IEEE Transactions on Software Engineering*, vol. 5, pp. 216–226, 1979.

[10] R. Albert, H. Jeong, and A.-L. Barabási, "Internet: Diameter of the World-Wide Web," *Nature*, vol. 401, pp. 130–131, 1999.

[11] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph Structure in the Web," in *Proceedings of the 9th International World Wide Web Conference (WWW '00)*, 2000, pp. 309–320.

[12] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On Power-Law Relationships of the Internet Topology," *ACM SIGCOMM Computer Communication Review*, vol. 29, pp. 251–262, 1999.

[13] J. R. Banavar, A. Maritan, and A. Rinaldo, "Size and form in efficient transportation networks," *Nature*, vol. 399, pp. 130–132, 1999.

[14] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a Social Network or a News Media?" in *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*, 2010, pp. 591–600.

[15] S. Milgram, "The Small World Problem," *Psychology Today*, vol. 2, pp. 60–67, 1967.

[16] "Google Search," http://google.com.

[17] "Bing Search," http://www.bing.com.

[18] "Yahoo! Search," http://www.yahoo.com.

[19] "Google Maps," https://maps.google.com/.

[20] "Bing Maps," http://www.bing.com/maps/.

[21] "MapQuest Maps," http://www.mapquest.com/.

[22] "Hitting It Off, Thanks to Algorithms of Love," http://tinyurl.com/yv4xep.

[23] "Inside Match.com: It's all about the algorithm," http://tinyurl.com/ko7aa89.

[24] "eHarmony's Algorithm of Love," http://tech.fortune.cnn.com/2010/09/23/the-algorithm-of-love/.

[25] "FlockDB," https://github.com/twitter/flockdb.

[26] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Power-Graph: Distributed Graph-Parallel Computation on Natural Graphs," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation(OSDI '12)*, 2012, pp. 17–30.

[27] 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 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*, 2010, pp. 135–146.

[28] S. N. Dorogovtsev and J. F. F. Mendes, "Evolution of Networks," June 2001, http://arxiv.org/abs/cond-mat/0106144.

[29] S. H. Strogatz, "Exploring Complex Networks," *Nature*, vol. 410, pp. 268–276, 2001.

[30] D. J. Watts and S. H. Strogatz, "Collective Dynamics of 'Small-World' Networks," *Nature*, vol. 393, pp. 440–442, 1998.

[31] A.-L. Barabási and R. Albert, "Emergence of Scaling in Random Networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[32] J. Kleinberg, "The Small-World Phenomenon: An Algorithmic Perspective," in *Proceedings of the 32nd ACM Symposium on Theory of Computing*, 2000, pp. 163–170.

[33] L. A. Adamic, R. M. Lukose1, A. R. Puniyani, and B. A. Huberman, "Search in Power-Law Networks," *Physical Review E*, vol. 64, 2001.

[34] R. Pastor-Satorras and A. Vespignani, "Epidemic Spreading in Scale-Free Networks," *Physical Review Letters*, vol. 86, pp. 3200–3203, 2001.

[35] M. Small, X. Xu, J. Zhou, J. Zhang, J. Sun, and J. an Lu, "Scale-free Networks which are Highly Assortative but not Small World," *Physical Review E*, vol. 77, 2008.

[36] L. A. N. Amaral, A. Scala, M. Barthélémy, and H. E. Stanley, "Classes of Small-World Networks," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 97, no. 21, pp. 111 490–11 152, 2000.

[37] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[38] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-Scale Graph computation on Just a PC," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, 2012, pp. 31–46.

[39] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford InfoLab, Technical Report 1999-66, 1999.

[40] S. Suri and S. Vassilvitskii, "Counting Triangles and the Curse of the Last Reducer," in *Proceedings of the 20th international Conference on World Wide Web (WWW '11)*, 2011, pp. 607–614.

[41] I. Bordino, P. Boldi, D. Donato, M. Santini, and S. Vigna, "Temporal Evolution of the UK Web," in *Proceedings of the 1st International Workshop on Analysis of Dynamic Networks (ICDM-ADN '08)*, 2008, pp. 909–918.

[42] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks," in *Proceedings of the 20th International Conference on World Wide Web (WWW '11*, 2011, pp. 587–596.

[43] P. Boldi and S. Vigna, "The WebGraph Framework I: Compression Techniques," in *Proceedings of the 13th International World Wide Web Conference (WWW '04)*, 2004, pp. 595–601.

[44] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing," in *Proceedings of the 8th European Conference on Computer Systems (EuroSys '13)*, 2013.

[45] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring Network Structure, Dynamics, and Function using NetworkX," in *Proceedings of the 7th Python in Science Conference (SciPy '08)*, 2008, pp. 11–15.

[46] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan, "Managing Large Graphs on Multi-Cores with Graph Awareness," in *Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC '12)*, 2012, pp. 41–52.

[47] "Apache Giraph," http://incubator.apache.org/giraph/.

[48] "GoldenOrb version 0.1.1," http://goldenorbos.org/.

[49] "Phoebus," https://github.com/xslogic/phoebus.

[50] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "GraphLab: A New Parallel Framework for Machine Learning," in *Proceeding of the 26th Conference on Uncertainty in Artificial Intelligence (UAI '10)*, 2010, pp. 340–349.

[51] "Apache Hama," http://hama.apache.org.

[52] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber, "Presto: Distributed Machine Learning and Graph Processing with Sparse Matrices," in *Proceedings of the 8th European Conference on Computer Systems (EuroSys '13)*, 2013.

[53] R. Power and J. Li, "Piccolo: Building Fast and Distributed Programs with Partitioned Tables," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, 2010, pp. 1–14.

[54] B. Shao, H. Wang, and Y. Li, "Trinity: A Distributed Graph Engine on a Memory Cloud," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD '13)*, 2013.

[55] D. Gregor and A. Lumsdaine, "The Parallel BGL: A Generic Library for Distributed Graph Computations," in *Proceedings of the 4th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC '05)*, 2005.

[56] J. Dean and S. Ghemawa, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Symposium on Opearting Systems Design and Implementation (OSDI '04)*, 2004, pp. 137–149.

[57] "Apache Hadoop," http://hadoop.apache.org/.

[58] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, 2012.

[59] "Apache Mahout," http://mahout.apache.org/.

[60] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient Iterative Data Processing on Large Clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[61] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A Runtime for Iterative MapReduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10)*, 2010, pp. 810–818.

[62] U. Kang and C. E. T. C. Faloutsos, "PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations," in *Proceedings of the 9th IEEE International Conference on Data Mining (ICDM '09)*, 2009, pp. 229–238.

[63] "InfiniteGraph," http://www.objectivity.com.

[64] "Neo4j," http://www.neo4j.org.

[65] A. Abou-Rjeili and G. Karypis, "Multilevel Algorithms for Partitioning Power-Law Graphs," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS '06)*, 2006.

[66] G. Karypis and V. Kumar, "Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs," in *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (SC '96)*, 1996.

[67] G. Karypis and V. Kumar, "Multilevel k-way Partitioning Scheme for Irregular Graphs," *Journal of Parallel and Distributed Computing*, vol. 48, pp. 96–129, 1998.

[68] I. Stanton and G. Kliot, "Streaming Graph Partitioning for Large Distributed Graphs," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '12)*, 2012, pp. 1222–1230.

[69] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez, "The Little Engine(s) that Could: Scaling Online Social Networks," in *Proceedings of the ACM SIGCOMM 2010*, 2010, pp. 375–386.

[70] S. Salihoglu and J. Widom, "GPS: A Graph Processing System," Stanford University, Technical Report, 2012.

[71] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li, "Improving Large Graph Processing on Partitioned Graphs in the Cloud," in *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC '12)*, 2012, pp. 1–13.

[72] M. K. Mckusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, vol. 2, pp. 181–197, 1984.

[73] C. K. Wong, "Minimizing Expected Head Movement in One-Dimensional and Two-Dimensional Mass Storage Systems," *ACM Computing Surveys*, vol. 12, pp. 167–178, June 1980.

[74] C. K. Wong, *Algorithmic Studies in Mass Storage Systems*. New York, NY, USA: W. H. Freeman & Co., 1983.

[75] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "BORG: Block-reORGanization for Self-optimizing Storage Systems," in *Proceedings of the 7th USENIX Conference on File and Storage Technologies*, 2009, pp. 183–196.

[76] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou, "C-Miner: Mining Block Correlations in Storage Systems," in *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies*, 2004, pp. 173–186.

[77] H. Huang, A. Hung, and K. G. Shin, "FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption," in *Proceedings of 20th ACM Symposium on Operating System Principles*. ACM Press, 2005, pp. 263–276.

[78] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, "On the Evolution of User Interaction in Facebook," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks*, August 2009.

[79] A. D. Brunelle, *blktrace User Guide*, February 2007.

[80] C. Mason, "Seekwatcher," http://oss.oracle.com/ mason/seekwatcher/.

[81] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending SSD Lifetimes with Disk-Based Write Caches," in *FAST '10*, 2010, pp. 101–114.

[82] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some Simplified NP-complete Problems," in *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, 1974, pp. 47–63.

[83] J. Petit, "Experiments on the Minimum Linear Arrangement Problem," *Journal of Experimental Algorithmics*, vol. 8, December 2003.

[84] J. Schindler and G. R. Ganger, "Automated Disk Drive Characterization," CMU-CS-99-176, Tech. Rep., 1999.

[85] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, vol. 20, pp. 359–392, 1998.

[86] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast Unfolding of Communities in Large Networks," *Journal of Statistical Mechanics: Theory and Experiment*, p. P10008, 2008.

[87] "Disk Allocation Viewer for Linux," http://davl.sourceforge.net/.

[88] "METIS: Family of Multilevel Partitioning Algorithms," http://glaros.dtc.umn.edu/gkhome/views/metis.

[89] R. Card, , T. Ts'o, and S. Tweedie, "Design and Implementation of the Second Extended Filesystem," in *Proceedings of the First Dutch International Symposium on Linux*, 1994.

[90] S. C. Tweedie, "Journaling the Linux ext2fs Filesystem," in *LinuxExpo*, 1998.

[91] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, vol. 10, pp. 26–52, February 1992.

[92] G. Ganger and M. F. Kaashoek, "Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files," in *Proceedings of the 1997 USENIX Technical Conference*, 1997, pp. 1–17.

[93] Y. Manolopoulos and J. G. Kollias, "Optimal Data Placement in Two-Headed Disk Systems," *BIT*, vol. 30, no. 2, pp. 216–219, 1990.

[94] P. Vongsathorn and S. D. Carson, "A System for Adaptive Disk Rearrangement," *Software: Practice and Experience*, vol. 20, pp. 225–242, March 1990.

[95] C. Ruemmler and J. Wilkes, "Disk Shuffling," HP Technical Report, HPL-91-156, Tech. Rep., 1991.

[96] S. Akyürek and K. Salem, "Adaptive Block Rearrangement," *ACM Transactions on Computer Systems*, vol. 13, pp. 89–121, May 1995.

[97] W. W. Hsu, A. J. Smith, and H. C. Young, "The Automatic Improvement of Locality in Storage Systems," *ACM Transactions on Computer Systems*, vol. 23, pp. 424–473, November 2005.

[98] B. Salmon, O. Salmon, E. Thereska, C. A. N. Soules, and G. R. Ganger, "A Two-Tiered Software Architecture for Automated Tuning of Disk Layouts," in *Proceedings of the First Workshop on Algorithms and Architectures for Self-Managing Systems*. ACM, 2003, pp. 13–18.

[99] J. A. Nugent, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau, "Controlling your PLACE in the File System With Gray-Box Techniques," in *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, 2003, pp. 311–324.

[100] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch," in *Proceedings of the 2007 USENIX Annual Technical Conference*, 2007, pp. 20:1–20:14.

[101] J. M. Pujol, V. Erramilli, and P. Rodriguez, "Divide and Conquer: Partitioning Online Social Networks," *CoRR*, vol. abs/0905.4918, 2009.

[102] J. M. Pujol, G. Siganos, V. Erramilli, and P. Rodriguez, "Scaling Online Social Networks without Pains," in *Proceeding of the 5th International Workshop on Networking Meets Databases*, October 2009.

[103] S. Fortunato, "Community Detection in Graphs," *Physics Reports*, vol. 486, no. 3-5, pp. 75–174, 2010.

[104] "GraphLab version 2.1," http://graphlab.org.

[105] "The Graph500 Benchmark," http://www.graph500.org.

[106] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida, "Characterizing User Behavior in Online Social Networks," in *Proceedings of the 9th ACM SIGCOMM Internet Measurement Conference*, ser. IMC '09. New York, NY, USA: ACM, 2009, pp. 49–62.

[107] "Facebook Growth," http://tinyurl.com/4flny8c.

[108] C. Cronin, "The Evolution of Storage – 2012 Isn't Just the Year of the Dragon," http://www.icc-usa.com/insights/the-evolution-of-storage-2012-isnt-just-the-year-of-the-dragon/, 2012.

[109] R. Branson, "Cassandra and Solid State Drives," http://www.slideshare.net/planetcassandra/cassandra-and-solid-state-drives-22509249, 2013.

[110] A. Lakshman and P. Malik, "Cassandra – A Decentralized Structured Storage System," *ACM SIGOPS Operating Systems Review*, vol. 44, pp. 35–40, 2010.

[111] "MongoDB," http://www.mongodb.org/.

[112] "Redis," http://redis.io.

[113] "Memcached," http://memcached.org.

[114] "Titan: Distributed Graph Database," http://thinkaurelius.github.io/titan/.

[115] "Redis-Graph: A powerful graph Implementation using redis sets," https://github.com/tblobaum/redis-graph.