A DISTRIBUTED METHOD FOR FAST FORCE-DIRECTED LAYOUT OF LARGE SCALE-FREE NETWORK GRAPHS

by

Nathan Lapierre

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
December 2015

To my parents, whose tremendous support has made this possible

Table of Contents

List of Tables

List of Figures

Abstract

Network graphs from Online Social Networks (OSNs) — representing network participants, and their relationships and interactions — are now readily available to researchers, and are an area of significant investigation. Visual layouts of these graphs are a common tool to help understand a network's underlying structure, form hypotheses, and communicate results. Force-directed placement is a layout method known to produce visualizations that contain a high level of useful information. However, producing an acceptable visualization can take an unreasonable amount of time, as OSNs can be very large, and existing layout techniques are computationally expensive.

In this work we propose a distributed computing method to speed up the completion time of social network graph layout. Using a commodity computer cluster, additional CPU resources can be allocated to the layout task. Our method builds upon existing graph layout techniques in the literature, and puts forward a graph partitioning scheme well-suited to graphs with small-world, scale-free network properties; properties that are naturally occurring in social networks.

We implement our method within the Hadoop distributed computing framework, and evaluate it for speedup and layout correctness on real-world social network graphs. Our results indicate that our method provides good speedup as computing resources are added, and that the resulting layouts correlate highly with their gold standards.

List of Abbreviations Used

API	Application Programming Interface
CPU	Central Processing Unit
FMM	Fast Multipole Method
GPU	Graphics Processing Unit
MDS	Multidimensional Scaling
OSN	Online Social Network
RAM	Random Access Memory
SFDP	Scalable Force Directed Placement Algorithm
SNA	Social Network Analysis
SNE	Stochastic Neighbor Embedding

Acknowledgements

I have many people to thank for making this thesis possible:

First and foremost, my co-supervisors, Dr. Gruzd and Dr. Matwin, for providing excellent support and feedback. Dr. Gruzd has been a great mentor for the many years that I have worked with him at the Social Media Lab. The other students at the lab have always been a great source of discussion and inspiration. As well, thanks to my readers, Dr. Smit and Dr. Blustein, for your insightful comments and questions.

An immense thank you to the many people who built the open source software that helps power my implementation, including Hadoop, Giraph, Impala, and DGA. Publishing your software as open source is very much appreciated.

Funding for lab equipment and infrastructure is from the Canada Foundation for Innovation fund awarded to Dr. Gruzd.

Finally, my family, friends, and partner have been resoundingly supportive during this degree, and I could not have finished it without them.

Chapter 1

Introduction

Online Social Networks (OSNs) are currently among the most popular websites on the Internet[3], connecting and facilitating communication for millions of people. A good deal of the public interactions between users of many of these websites can be freely obtained in machine-readable format through an API. Researchers can analyze these data to learn about specific social structures, or make discoveries about broader social phenomena[58]. A research method commonly used for this work is Social Network Analysis (SNA). SNA is concerned with understanding social relationships through network analysis and graph theory[28]. Social networks are represented as a graph structure, with vertices as the members of the network and edges as their relationships. Graph properties, such as degree distribution, or metrics calculated through graph traversal, such as centrality, offer insight into how the social network is organized. SNA may help us discover important network members, communities, communication channels, or many other social occurrences relevant to "social psychology, social anthropology, communication science, organizational science, economics, geography and, especially, sociology."[28]

Social network graphs are commonly visualized to aid in analysis. A diagram of the network's actors and their relationships (also known as a sociogram[18]) can help reveal information about the network's structure that may be difficult to determine through quantitative methods alone. These visualizations are useful to "form hypotheses and communicate results."[18] Figure 1.1 is an example of a visualization of a Facebook friendship network (NB: the terms 'visualization' and 'drawing' are both used in the literature, and likewise used interchangeably within this thesis). Nodes represent Facebook users, with edges between them representing friendships. Closer node distances within the diagram represent stronger relationships. This drawing reveals a network with eight densely connected clusters, each relatively weakly interconnected. The node's color helps to distinguish its cluster membership. Visualizations can be used as starting point to quickly form a hypothesis about the network's social structure; for example, clusters may represent people with varying political ideologies, geographic localities, group memberships, or other properties.

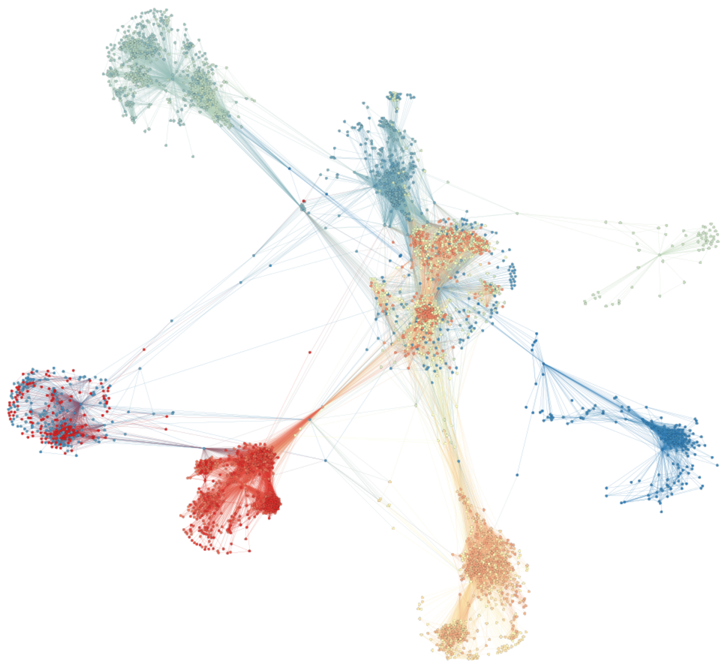

Figure 1.1: An example of a social network visualization, representing Facebook friendships. The particular layout algorithm used reveals eight distinct clusters of users, each weakly interconnected. This figure was generated using Gephi[7] software, with Facebook network data from McAuley & Leskovec[1].

Many different algorithms and techniques, implemented in a variety of software packages, are available to visualize social networks[14]. Often different techniques produce visualizations that can vary significantly in appearance, optimized for what its author considers to be the best layout for certain tasks[23]. For example, one algorithm may attempt to reduce overlap of large clusters, while another may work at a smaller scale and aim to reduce node overlap or edge crossings. Most visualization techniques used with OSNs employ a variant of force-directed graph layout[18]. The positions of nodes in the drawing are determined by applying attracting forces between edges and repelling forces between nodes[24].

1.1 Problem

While drawing social graphs can be useful, many existing techniques are not well-suited to visualizing very large networks. Most cannot visualize networks larger than several tens of thousands of nodes on typical commodity hardware. Many OSNs now have hundreds of millions of active users. It is relatively easy for researchers to obtain network datasets containing millions, or even billions of nodes and edges, but it is currently problematic or impossible to visualize them.

Naïve force-directed graph drawing has a high run time complexity of $O(|V|^3 + |E|)$, where the work is dominated by the calculation of forces between each pair of vertices. Extensive prior work has been done to overcome various technical and algorithmic speed limitations of graph layout techniques. Algorithms exist with considerably reduced time complexity, some taking advantage of parallel computing on multi-core computers, which helps reduce the total time needed to draw a graph. However, there are currently gaps in prior work to bring graph layout to distributed computing architectures. Distributed algorithms allow the size of the problem to scale beyond the CPU and memory bounds of a single machine, simply by adding additional computers to a networked cluster as the size of the work grows. Prior work may be limited in this area for several reasons: graph problems are difficult to translate to a distributed environment, as we will see in the next Chapter; and, researcher access to large social network data and distributed computing is a recent occurrence.

1.2 Contributions

This project presents a new distributed algorithm and software implementation that allows for the visualization of much larger networks than is possible using existing, non-distributed methods. The implementation is built on Hadoop, a popular framework that eases the management and programming of distributed clusters[32]. By leveraging the unique properties of social networks in particular, a good graph partitioning can be obtained, reducing communication overhead that can otherwise dramatically limit performance in a distributed environment. With this graph partitioning technique, and building on existing layout techniques from the literature, visualizations of very large networks can be completed in a more reasonable time by scaling the available computing resources.

The primary contributions of this work are:

1. A graph partitioning scheme well-suited to social graphs; and
2. A multilevel distributed layout technique, building upon prior work, leveraging this graph partitioning.

In evaluating this work, two key questions arise:

1. Does the distributed algorithm and implementation perform and scale well?
2. Is the resulting visualization correct?

Therefore, two evaluations were undertaken. First, a speedup test was performed to measure graph drawing time and scalability, controlling for a variety of real-world social graph inputs and distributed cluster conditions. Second, layout correctness was evaluated by sampling Euclidean distances between pairs of nodes, and testing for correlation between the layout distances generated by this distributed system, and the original layout algorithm upon which this work is based. Given the nature of this work, evaluation of the claims is empirical.

Results of our evaluation show that our method achieves speedup on our cluster, and that layout results correlate with the non-distributed version across our test datasets.

1.3 Thesis Structure

The remainder of this thesis document contains 4 chapters, as follows. In Chapter 2, we will review prior work necessary to understand graph drawing, social network graph structure, and distributed computing as they apply to visualizing large social networks. We will detail several specific works that will become the foundation of our system design. Chapter 3 will describe the algorithmic and implementation design of our system. We will present our evaluation methodologies and results in Chapter 4.

Finally, we will conclude this thesis in Chapter 5, with a discussion on the system's implications and possibilities for future work.

Chapter 2

Background

Graph drawing is a wide-reaching field that has produced a significant body of literature. Graph drawing deals with the automated layout of graph data structures. Graphs serve an enormous breadth of application domains; drawing them can, according to Sugiyama[50], help solve "extremely difficult problems in human cognition..." The *Handbook of Graph Drawing and Visualization*[52] explores graph visualization's use in understanding "biological networks, computer security, data analytics, education, computer networks, and social networks."

While graph drawing is relevant and useful to many areas, this work is only interested in drawing social network graphs. Social networks exhibit several unique properties that we will attempt to leverage to develop a scalable distributed drawing algorithm. We will review these properties first in this chapter. Unless otherwise specified, any further references to graphs or networks will imply social networks with these characteristics.

Force-directed layout methods are most commonly used to draw social network graphs; we will review the number of advantages they provide over other techniques. As force-directed layout is a computationally expensive task, we will next assess some of the existing techniques to improve its run-time performance. We will also briefly explore alternative techniques to improve the drawing of large graphs. We will justify why we also want to be able to draw large graphs using conventional force-directed visualization methods.

Finally, we will examine work relevant to speeding up or scaling up the drawing of large graphs through distributed computing. Distributed computing allows the reduction of processing time, or the size of the data to grow as necessary, by adding resources to a networked, distributed computer cluster. While this can be useful to layout large graphs, our investigation finds that there has been little work in this area so far.

2.1 Social Network Graphs

Naturally forming human social networks exhibit several unique graph properties[19]. At a high level we can think of these graphs as appearing sparse, but with densely connected local clusters, similar to the network in Figure 1.1 in the previous chapter. We will explore two concepts to formally understand this: the ‘small-world’ network, and the ‘scale-free’ network.

2.1.1 Small-world Networks

Watts and Strogatz[59] describe the connection topology of small world networks, of which social networks are a subset, as “somewhere between” completely regular and completely random. They are highly clustered, but also contain a percentage of random edges, or short-cuts, between distant nodes. This was first demonstrated by Travers and Milgram’s 1969 “small world” experiment[55], giving networks with this property their name. Figure 2.1 shows how the Watts and Strogatz model can be applied to a regular, clustered network to produce a small world network. In a human social network, the demonstrated random edge rewirings occur naturally to varying degrees.

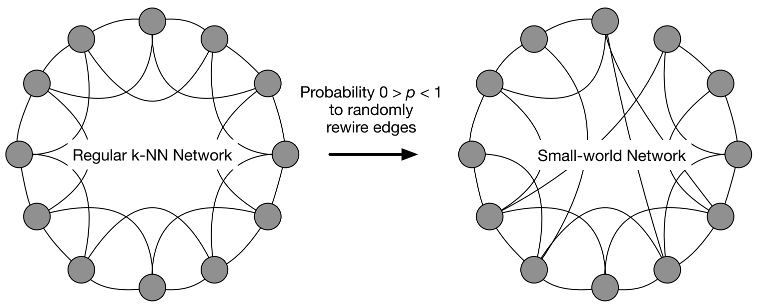

Figure 2.1: Starting with a regular network where nodes are connected to their k nearest neighbors (here $k = 4$), a small-world network can be created by randomly rewiring edges with probability $0 < p < 1$. When $p = 1$, the network becomes a fully random network. Adapted from Watts and Strogatz[59].

Small-world networks can be quantified by two properties: characteristic path length, L, and clustering coefficient, C. L, a global property, indicates the average of the shortest path lengths between each pair of vertices in the graph. In a small world network, the value of L drops compared to a regular network, as shortcuts are introduced. C is a local property representing the cliquishness of a neighborhood. For each vertex, v, C represents the ratio of total possible edges of v's neighbors, versus the number of edges that actually exist. When $C = 1$ all neighbors of v have edges between them. For example, in a social friendship network, C will be high when most of your friends know each other. In small world networks, the graph's average clustering coefficient is much greater than in a random network.

2.1.2 Scale-free Networks

Social networks also exhibit unique degree distribution properties, known as a scale-free network. Scale-free networks, described by Amaral et al.[4], have a degree distribution that follows a power law decay. That is, many vertices in a scale-free network have few connections, while few vertices are very highly connected. Figure 2.2 demonstrates this distribution in a real-world social network. Figure 2.3 visualizes a typical scale-free network structure, where several "hubs"[58] are much better connected than other nodes.

A scale-free network can also be understood by a preferential attachment model. Vertices in a growing network are more likely to connect to nodes that are already highly connected.

Small-world, scale-free properties have been shown to appear in very large OSNs. Work examining MySpace and other websites in [36] and [45], while limited at the time to relatively small scale data collection and analysis, showed them to be small-world, scale-free networks. In 2011, Facebook's entire network graph, containing over 720 million vertices and 68.7 billion edges, was analyzed and shown to have these structural characteristics[56]. Bakhshandeh and Samadi's[5] analysis of Twitter showed a small-world network with a characteristic path length of 3.43.

We can take advantage of this structure of social networks — many large, dense clusters that are relatively weakly interconnected — as a starting point to divide our layout workload. We will develop this idea more deeply in the coming chapters.

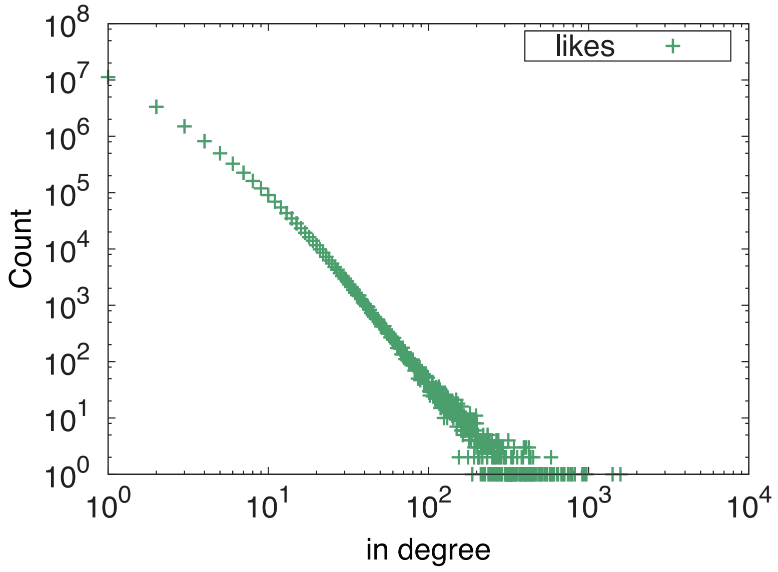

Figure 2.2: An example of a social network, 'likes' on Instagram photos, exhibiting a scale-free in-degree distribution. Data was collected from the Instagram public API. **Methodology**: A 'likes' network — where edges indicate that a user likes another user's photo — was constructed by collecting data from Instagram's `/users/`*`user-id`*`/media/recent` API endpoint, where *user-id* is randomly sampled (*user-id*s are sequential). The in-degree distribution of the resulting network was calculated with Pegasus[34].

2.2 Graph Drawing

The essential goal of graph drawing is to embed a graph structure into a space — typically a 2D plane, sometimes a 3D space — such that the resulting diagram conveys a high level of information about the graph. Sugiyama argues that "a *good* drawing gives a high possibility of quickly and accurately communicating the meaning of a diagram, but a *bad* drawing gives a high possibility of confusion

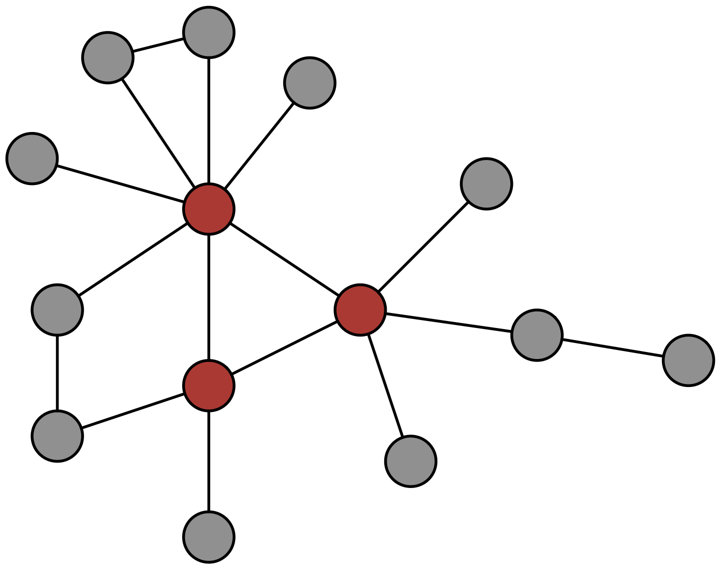

Figure 2.3: This visualization shows a typical structure of a scale-free network. The three red nodes have the highest degree, and are the ‘hubs’ of the network.

or misunderstanding.”[50] However, the criteria for good layout embedding varies widely.

Social networks graphs are commonly drawn using a class of algorithms called force-directed layout (also known as spring embedders)[18]. Force-directed layout involves applying attracting forces between graph edges, and repelling forces between nodes. Figure 2.4 illustrates these forces. Graphs drawn this way naturally tend to reduce overlap, appear symmetrical, and are overall “aesthetically pleasing.”[24]

Other techniques to draw graphs do exist, such as dimensionality reduction approaches including t-SNE[21] (*t-Distributed Stochastic Neighbor Embedding*), and MDS[10] (*Multidimensional Scaling*). While both have been applied with success to drawing social network graphs, they do have disadvantages. t-SNE tends to get trapped in local minima, while MDS can have difficulty in reaching a stable, globally optimal layout. That is to say, these techniques do not produce a layout as precise as force-directed with possibly negative effects on how a graph visualization is interpreted. Therefore, we contend that improving force-directed drawing of large social networks remains worthwhile.

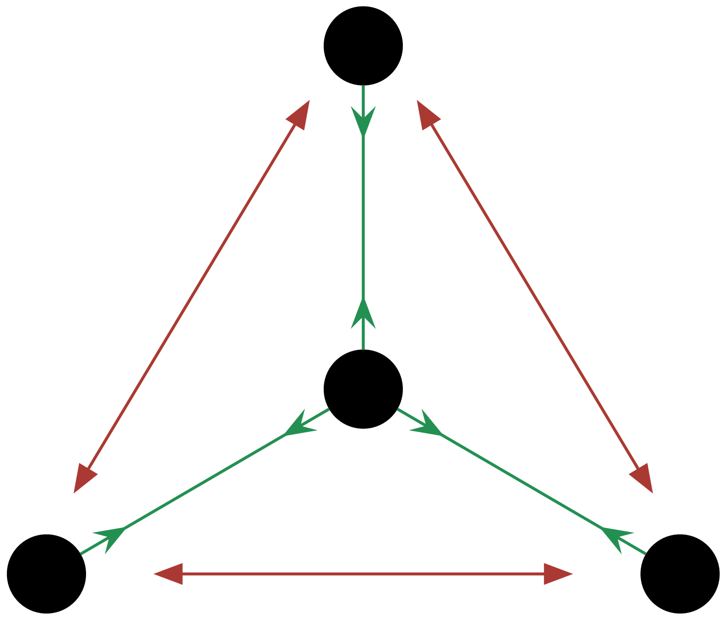

Figure 2.4: This diagram shows the attracting and repelling forces applied to a small graph during force-directed layout. The edges (green) attract, while the nodes repel (forces shown in red). The resulting graph layout is naturally symmetrical.

The large variety of force-directed implementations differ in their specific calculation of attracting and repelling forces. The calculations are typically based on a physical metaphor; for example, spring energy (Hooke's law), or electrical charge (Coulomb's law). Several force-direct layout algorithms in popular use include: Fruchterman-Reingold[20], ForceAtlas2[31], LGL[2], and SFDP[30].

Each version strives for different aesthetics. Fructerman-Reingold aims to generate drawings with uniform edge lengths. It is the earliest work listed, and may not be well-suited to meaningfully representing very large graphs. ForceAtlas2 provides more advanced layout heuristics, and several optimizations for faster layout of larger graphs than Fruchterman-Reingold. LGL (Large Graph Layout) is designed to draw larger graphs, in particular to emphasize dense clusters. Finally, SFDP (Scalable Force-directed Placement) provides the most advanced set of layout heuristics of these methods, that are particularly well-suited to drawing large graphs. SFDP will serve as a basis of this work, and we will examine its details more closely in Chapter 3.

2.2.1 Force-directed Drawing Improvements

Since unoptimized force-directed layout must calculate the interaction forces for every edge and between every pair of nodes, a significant aspect of designing these algorithms is to reduce their run-time complexity. Given that nodes with a large distance between them will produce negligible interaction forces, as these forces often decay quickly, many algorithms reduce complexity by ignoring these forces.

Fruchterman-Reingold provides an option to partition the input graph into a simple, equivalently sized 'grid'. Nodes that lie outside the neighboring grid squares are rejected during the force computation. This approach improves performance only for graphs of uniform distribution, therefore excluding many real-world networks.

Partitioning the graph to allow better balancing for arbitrary graphs is possible through techniques such as Fast Multipole Method (FMM)[61], and Barnes-Hut Oct-tree[6]. Barnes-Hut optimization is used in ForceAtlas2 and SFDP. Similar to Fruchterman-Reingold, Barnes-Hut divides the graph space into cells, however each of these cells is then "recursively divided into eight subcells whenever more than one particle is found to occupy the same cell."[6] The resulting partitioning, suitable to a variety of graph structures, can be seen in Figure 2.5.

Another optimization category is multilevel layout[24]. By simplifying the graph during initial layout, large groups of vertices can be moved at once, rather than calculating forces for each individually. SFDP employs this alongside Barnes-Hut. The graph is first coarsened by collapsing pairs of adjacent vertices. The resulting simplified graph, often comprising $\sim$50% fewer vertices, is laid out. With better than random starting positions from this grouped layout, individual vertex positions can now be calculated with a reduced cost. Overall this approach allows the full graph can be laid out more quickly and with higher quality.

Some implementations of SFDP and several other algorithms reduce processing time through parallel computing. Calculation of node forces can be done in parallel on mutli-core CPUs or GPUs. This works well on shared-memory computers due to relatively low main memory or cache access times. Concurrent calculations requiring a piece of shared data can access it with little performance penalty. This does not translate well to distributed computer architectures. Concurrent calculations may require the same data in memory, but exist on separate machines within the cluster.

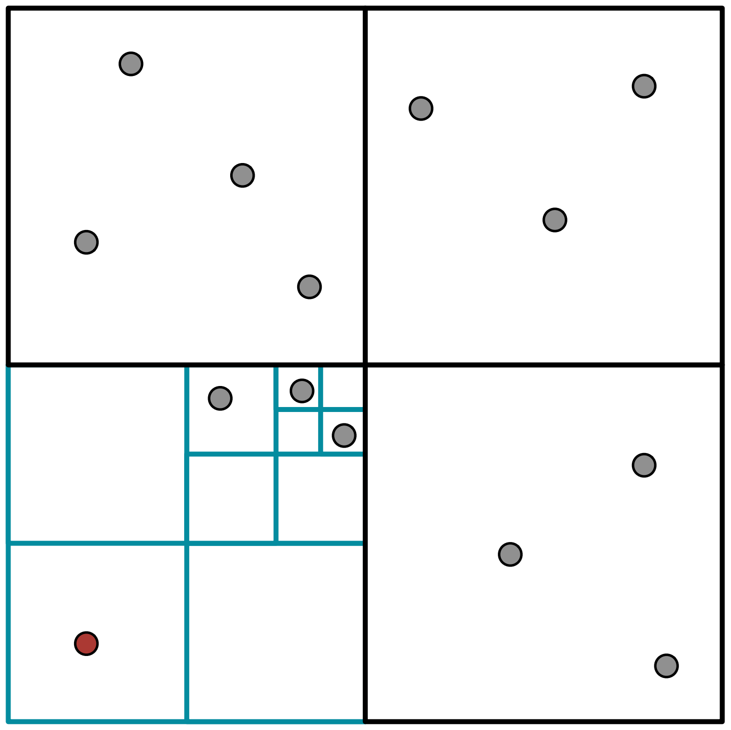

Figure 2.5: A graph with a number of vertices are shown. The red vertex is under consideration to calculate attracting and repelling forces for force-directed layout. The graph is partitioned by Barnes-Hut octree (blue outline). Distant particles in the three large surrounding quadrants can be ignored as their force effect is negligible. The force from the vertex in the second level of the tree must be calculated, while the third level vertices can be approximated together. Overall the number of calculations is reduced considerably. Summarized from [16].

A network request for this data is several orders of magnitude slower than a request to main memory or cache. Therefore, a large challenge of distributed algorithm design, and of this thesis work, is to reduce or eliminate network communication overhead.

2.2.2 Visualization of Large Graphs

Beyond computational difficulties, drawing large graphs poses other challenges.

With a high number of nodes and edges to draw, a visualization of a large network may become incomprehensible. Images that become too cluttered may lose their meaning. This is known as a “hairball”[23] drawing; an example is shown in Figure 2.6. Edge-bundling[29] can help to reduce visual clutter by combining overlapping edges into thicker lines to represent the relative strength of the relationships between node clusters. Likewise, clusters of similar nodes can be visually combined. More novel node representation techniques, such as a Nodetrix[27], have also been developed. As these methods involve some loss of information, the effect on a researcher's ability to interpret a visualization is unclear.

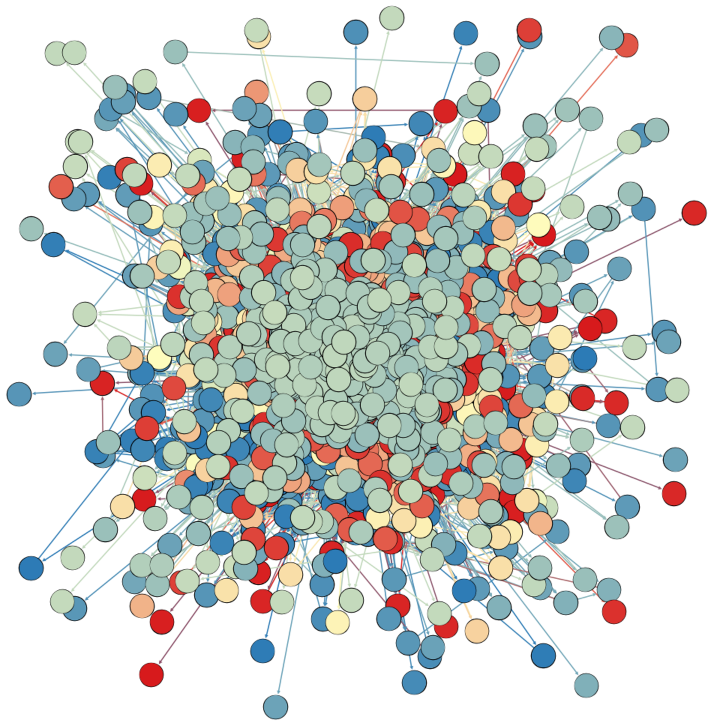

Figure 2.6: This is the same Facebook network of Figure 1.1, but drawn using a force-directed layout algorithm with parameters that are not optimal for a network of this size. The nodes and edges overlap to create a "hairball" that is no longer useful toward understanding the network structure.

Instead, visualizing a smaller sample may appear to be a viable solution to working with intractably large networks. However, this poses its own set of difficulties. As shown by Stumpf et al.[49], "subnets of scale-free networks are not scale-free." That is, taking a random sample of a scale-free social network will give a sample that lacks the structure of its full graph. The degree distribution of the sample will not be scale-free. A visualization of this sample may therefore be inaccurate or misleading. Even where the researcher is aware of this problem, properly taking a sample can be very difficult, according to Stumpf et al.: "unless sampling reverses the sequence of events by which networks were generated, the subnet will not have a scale-free distribution."

We believe this work towards scalably drawing complete, large social networks is therefore useful. Where force-directed drawing of large networks can be done while maintaining comprehensibility, we can avoid current challenges of potential loss of detail with alternative drawing techniques, or possibly troublesome sampling methods.

2.3 Distributed Computing

In general there are three unique challenges to parallelizing, and in turn distributing graph algorithms, per Lumsdaine et al.[41], summarized here:

- **Data-driven:** Graph problems are typically data-driven. It is often not possible to divide graph work up by partitioning sections of the code itself across the cluster.
- **Unstructured/poor locality:** General graphs often represent vertices with relationships that are unstructured. It is difficult to divide an arbitrary graph up to ensure balanced workload and good locality (where the graph partition resides primarily in the same computer where the work takes place).
- **High data access overhead:** Graph problems often involve much traversal, resulting in high memory access or communication overhead that can be a challenge to reduce.

To help address this complexity, Malewicz et al. introduced the *Pregel* programming model[42]. It is designed as a distributed, parallel system specifically for graph work. During a "superstep", for a partition of the graph, each parallel worker performs calculations for a single vertex. Messages are passed at the end of each superstep, during a synchronization step, with any changes to the graph's state. The programmer only needs to implement a single compute() method, and is exposed only to a view of a single vertex, its neighbours, and any incoming messages. Pregel relieves the programmer of workload balancing, implements data-driven computation, and helps to minimize communication overhead.

The Pregel model has been implemented in a number of open source software packages[25], such as GraphLab, GraphX, and Giraph[11]. Giraph has the appealing feature of being implemented on the Hadoop distributed computing platform.

Hadoop provides a distributed file system, and handles distributed workload assignment and many other aspects of distributed cluster management. As Hadoop provides an extensive toolkit, is in widespread use, and accessible to researchers[26], we will be working with the Hadoop and Giraph frameworks in this work. Further details on these frameworks will be presented in the Method chapter.

2.3.1 Prior Distributed Graph Drawing Work

We were able to find two relevant papers describing methods to bring force-directed graph layout to parallel or distributed memory architectures; work by Tikhonova & Ma[53] in 2008, and more recent work by Yunis et al. in 2014[61]. While these papers share some foundations with our method, they do differ in important ways.

Tikhonova & Ma[53] assign vertices to processors by their degree, to help achieve better data locality. High degree vertices are grouped together, and a distributed, but otherwise standard Fruchterman-Reingold layout is performed. This provides a coarse graph layout, which is iteratively improved by adding groups of lower degree vertices.

This method was designed for Cray supercomputer systems; machines that provide extremely high-speed interconnections (50+ Gbps) within their clusters. Consequently, the authors seemed less concerned about inter-machine communication overhead. Our method, designed to minimize inter-machine communication, should be better suited to a Hadoop environment typically composed of commodity hardware with much slower, Ethernet-based interconnections.

The more recent work by Yunis et al. implements a graph partitioning based on Fast Multipole Method (FMM). FMM is similar in principle, but more sophisticated than the Barnes-Hut method previously described. The authors claim good speedup and performance on single CPUs and GPUs. While their library will run on distributed clusters, this was not evaluated, and they warn that process communication "takes a significant amount of time," suggesting that it would not immediately translate well to a distributed memory environment.

Finally, unlike our method, neither previous algorithm is specifically designed to draw scale-free, small-world networks, including social networks. Tikhonova & Ma suggest "Additional structural information about a graph is sometimes known in advance. . . . With this knowledge, an algorithm designed to perform well on a specific type of graph can be used to produce more aesthetically pleasing layouts and result in better running times.”

Chapter 3

Method

Our method is built on two key concepts. First, that social networks exhibit properties that may allow them to be well partitioned for certain tasks. Second, that we can use this partitioning to achieve good distributed speedup for a multilevel force-directed layout method.

This section will first formally describe the algorithms of our method. Next we will explain how we implemented these algorithms using a number of distributed graph processing technologies.

3.1 Algorithms

Our overall method is described in Algorithm 1, below. Referenced within Algorithm 1 are two additional algorithms that we will subsequently look at in further detail.

The algorithms below indicate their input and output conditions, the procedure's steps in pseudocode, and, where applicable, indications of any steps to be run in parallel or across a distributed cluster.

Algorithm 1 Distributed force-directed graph layout

```
Input: A graph, $G = (V, E)$
Output: Coordinate vectors $(x, y) = \{x_i, y_i | i \in V\}$
1: $C \leftarrow$ distributedLouvainModularity($G$) {Algorithm 2}
   Each cluster node runs the following, with $C$ shared across all nodes:
2: for community subgraphs $c$ in $C$ do
3:     if $|c| >$ Top 20% then
4:         $layout_c \leftarrow$ parallelSFDP($c$)
           {20% value was found empirically — see algo. description}
5:     else
6:         $layout_c \leftarrow$ random initial layout of $c$
7:     end if
8: end for
   Master node runs the following:
9: $(x, y) \leftarrow$ parallelSFDP($layout_C$)
```

Algorithm 2 Distributed Louvain modularity community detection

```
Input: A graph, $G = (V, E)$
Output: Community memberships $C = \{C_i | i \in V\}$
 1: $C \leftarrow$ Random initial communities
 2: while vertices changing communities $> 0$ do
 3:    for vertex $v \in V$ do
 4:       $\Delta Q_{prev} = 0$
 5:       for all neighboring communities $c(v)$ do
 6:          $Q = modularity(v, c)$ {See Equation 3.1}
 7:          if $\Delta Q > \Delta Q_{prev}$ then
 8:             $v_c = c$ {Change communities if resulting modularity is increased}
 9:             $\Delta Q_{prev} = \Delta Q$
10:          end if
11:       end for
12:    end for
13:    $G = G$, compressed to new communities
14: end while
```

Algorithm 3 Parallel Scalable Force-Directed Placement (SFDP)

```
Input: A graph, $G = (V, E)$, initial coordinates $x = \{x_i | i \in V\}$, and maximum
    iterations, $m$
Output: Coordinate vectors $x = \{x_i | i \in V\}$
 1: while not converged and iterations $< m$ do
 2:   for vertex $v \in V$ do
 3:     $Q$ = Barnes-Hut Tree of $v$'s neighbors
 4:     Calculate new coordinates for $x$:
 5:     {Repulsive forces of close vertices} (Electric model)
 6:     {Attractive forces of $v$'s edges} (Spring model)
 7:     {Inter-group attractive forces (depth of $Q >$ empirical 'far' distance)}
 8:     {Inter-group repulsive forces}
 9:     {Intra-group attractive forces}
10:     Calculate $\Delta$energy (adaptive cooling model)
11:     if $\Delta$energy $= 0$ then
12:       converged = true
13:     end if
14:   end for
15: end while
```

Algorithm 1 takes a graph as input, and produces 2D coordinates for each vertex.

Step 1 assigns each vertex a community through a distributed Louvain modularity process[8], distributedLouvainModularity(), described in Algorithm 2. The algorithm as it is described is identical to a non-distributed Louvain modularity. It is only the implementation within a distributed graph framework, described later, that allows this algorithm to run in parallel in a distributed computing cluster. Nonetheless, we chose to specify the algorithm as "distributed" Louvain modularity as it is very important to run this stage over the distributed cluster in order to attain good speedup overall.

Next, the largest of these communities are laid out with parallelSFDP(). Our criterion to determine the largest communities is to take the top 20% by size. As community size also follows a power-law distribution, we have found that the top 20% of communities are likely to contain the majority of the graph's nodes. SFDP, Scalable Force Directed Placement, is an optimized force-directed layout algorithm by Hu[30], and described in Algorithm 3 in a greatly simplified form. It runs on mutli-core computers, with vertex pair force calculations computed in parallel. In our distributed version, multiple communities are laid out in parallel across the cluster. Communities that fall below the specified size threshold, including isolated vertices, are assigned random initial positions.

All position information is now aggregated to a single machine. parallelSFDP() is run again on the entire graph, seeded with this position data.

The key concept behind our method is that by seeding the final layout step with starting coordinates that are much better than random, the number of iterations required to achieve global layout stability is greatly reduced.

Despite the added overhead of pre-processing the graph with Louvain modularity, and calculating the layout more than once, speedup can still attained by laying out the graph's major communities in parallel across the distributed cluster (when sufficient graph data and computing resources are available).

3.2 System Implementation

The system is built on a number of recent technologies that have been developed to ease the creation of distributed graph tasks. Here we will look at the technical challenges we faced, explore technologies designed to help overcome these challenges, and describe how we use these technologies to implement our system.

3.2.1 System Frameworks

As discussed in the Background chapter, designing distributed algorithms for graphs is more difficult than for many other data structures. Programming these algorithms pose their own challenges as well.

Developing any distributed system requires careful consideration of many well studied problems, including communication, coordination, workload balancing, and others. One software framework built to address many of these problems is Hadoop. The Giraph framework further extends Hadoop to facilitate distributed graph processing. These two frameworks form the foundation of our implementation.

Framework 1: Hadoop

Hadoop is a collection of open source software packages, primarily written in Java, to help make the creation of distributed systems and software easier[32]. A Hadoop cluster automatically and transparently handles communication, coordination and workload management, graceful entry and failure of nodes in the cluster, and distribution and replication of data across the cluster. This leaves the programmer free to focus on the task of implementing their distributed algorithm.

The creation of Hadoop was inspired by Dean & Ghemawat's MapReduce[15]. MapReduce is a programming paradigm in which a programmer implements only two methods: *Map()*, and *Reduce()*. These two steps can model a wide variety of distributed algorithms. *Map()* processes key/value pairs within the data, in parallel, creating intermediate values. Intermediate values are *shuffled* and transparently distributed to reducers. *Reduce()* merges all intermediate values of the same key. The concept can be thought of similarly to a massively parallel divide and conquer algorithm. MapReduce is implemented in Hadoop, with various improvements, under the name YARN, which also encompasses the task scheduling system.

Hadoop Distributed File System (HDFS) handles file replication and distribution across the cluster. Files added to HDFS are automatically split into 'chunks' and distributed to the cluster's *datanodes*. Redundancy can (optionally) be achieved through chunk replication. The cluster's *namenode* manages access and references to files. HDFS is accessed through the command line in much the same way as a standard unix file system, or through Hadoop-based programming frameworks through built-in APIs.

Hive and Impala are software packages used in this work that also fall under the Hadoop umbrella. Hive allows structured files (e.g. CSV or TSV files) residing in HDFS to be represented and queried as though they were relational database tables. Queries are written in HiveQL, an extensive subset of standard SQL. Queries are translated into MapReduce jobs and run across the distributed cluster. Hive allows the programmer to quickly implement otherwise complicated MapReduce tasks in a familiar and concise query language. Impala is a further extension of Hive, using Hive tables, but bypassing MapReduce all together to implement a custom-made, and much faster query engine. Impala/Hive tables are used throughout our implementation to manage data between the algorithm's steps.

Framework 2: Giraph

Hadoop provides many advantages to distributed systems builders. However, it is not well-suited to working with graph data. MapReduce is a slow and difficult framework in which to implement graph algorithms[33]. The Giraph framework was created to help resolve these issues through two primary means: a new programming and data processing model, and HDFS-based graph data storage and representation.

Giraph's programming and processing model is an implementation of Pregel[42], a scalable graph processing infrastructure that is in turn based on the well-known Bulk Synchronous Parallel (BSP) model[22]. The programmer only needs to implement a single *compute()* method, and is exposed only to a view of a single vertex, its neighboring edges, and any incoming messages. During a "superstep", for a partition of the graph, each parallel worker performs calculations for a single vertex, based on its view of neighboring edges, or data in any incoming messages. Messages are passed at the end of each superstep with any changes to the graph's state. Pregel/Giraph relieves the programmer of complicated communication overhead and workload balancing.

Giraph provides the functionality to automatically load graphs from HDFS into memory from several predefined formats, including edge lists and adjacency lists, or through a custom format specification.

3.2.2 System Steps

Figure 3.1 shows a high-level diagram of the system's components. From input of the graph edge list, the flow of our method is followed through the four numbered steps in the diagram. For each step, described in detail in the sections below, parentheses indicate the technologies used to implement it. Steps take place either over a distributed cluster, or on a single computer, as specified. Finally, our method ends with the output of a completed network visualization.

Step 1: Graph Partitioning

The first stage of our method is to pre-process the graph. As we have established, social network graphs typically exhibit small-world, scale-free properties. In practical

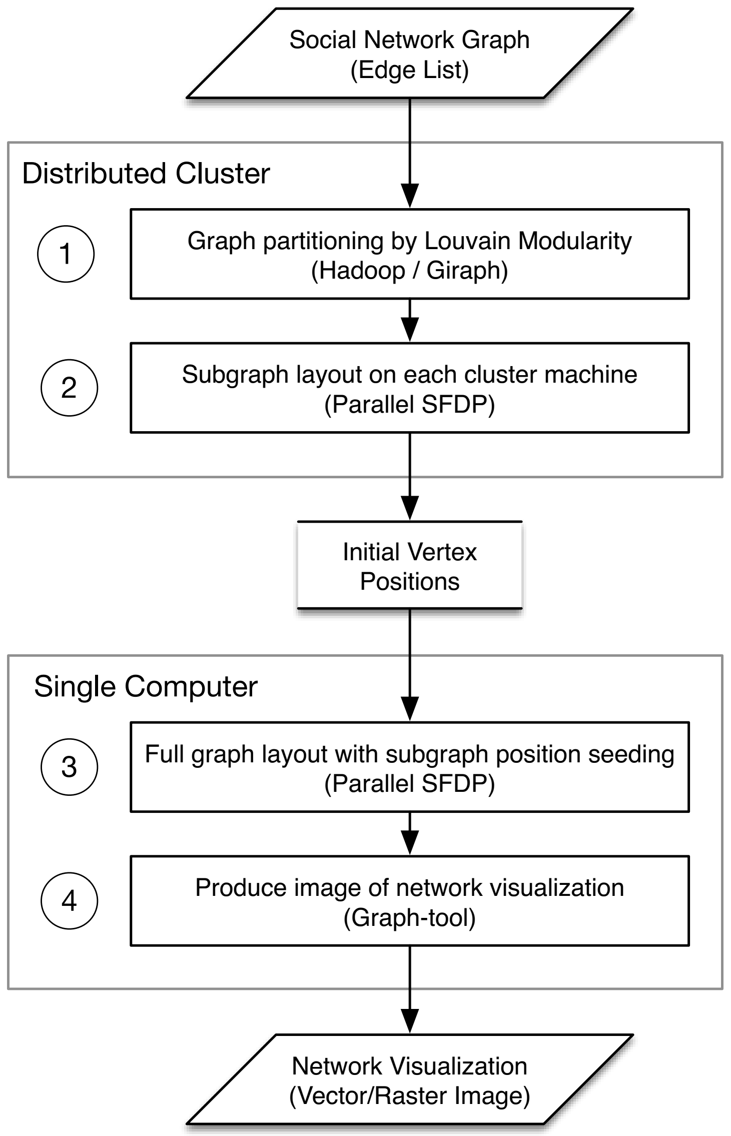

Figure 3.1: This diagram shows the flow of steps taken by our method to produce a social network visualization from a graph input. Technologies used to implement each step are shown in parentheses.

terms, they consist of many densely connected clusters that are relatively weakly interconnected. Partitioning the graph into these clusters cuts far fewer edges than a random or uniform partitioning.

To detect these clusters we can choose from many existing algorithms in the *community detection* literature; Fortunato provides an extensive survey[17].

Ultimately we chose Louvain modularity by Blondel et al.[8], for a variety of reasons:

- **Fast**: Foremost in our choice, Louvain modularity is very fast. Its expected run-time is $O(|V|log|V|)$.
- **Unsupervised**: Unlike other algorithms, Louvain does not require the number of communities within a network to be known in advance, which would be infeasible for arbitrary graphs.
- **Non-overlapping**: While a vertex in a real-world graph may belong to more than one community, Louvain presents non-overlapping communities. This fits well with our use case of simple graph partitioning.
- **Low resolution:** Modularity has a tendency to create large communities. This is ideal in our case, as fewer communities to process in later steps of our method will decrease distributed run-time overhead.

Louvain modularity does have weaknesses, primarily its instability. The detected communities in a graph vary over iterations of the algorithm. Modularity is a greedy, randomized algorithm. Vertices may be placed into a community due only to randomness. This can account for our method's variance in layout correctness, as shown in the following chapter.

Nonetheless, we believe Louvain's speed advantage outweighs any negative impact on layout correctness, as detailed in the Evaluation Chapter. The Future Work section briefly discusses further investigation of the impact the choice of community detection algorithm has on layout results.

The Louvain method is an optimization of general modularity-based community detection, where modularity (Q) is defined in Equation 3.1 as:

$$Q = \frac{1}{2m}\Sigma_{ij}\left[A_{ij} - \frac{k_i k_j}{2m}\right]\delta(c_i, c_j) \tag{3.1}$$

Q is a value between -1 and 1, comparing the the density of the edges within a community to those outside of it. A_{ij} is the edge weight between nodes i, j. m is the number of edges in the graph. k is the node's degree. $\delta(c_i, c_j)$ is the Dirac delta function[35] of communities, c. Optimizing for higher modularity places vertices in their most well clustered communities.

Louvain modularity begins by assigning each vertex its own random community. At each iteration a community may move to a neighboring community with a higher modularity value. After all communities have been updated the graph is compressed to represent each community as a single vertex. The stages repeat until no further modularity improvements are attained.

Our method runs Louvain modularity over a distributed cluster, providing greater speedup and scalability. The general algorithm is described above in Algorithm 2, and is implemented in Giraph and MapReduce. This implementation uses a subset of previous open source work in Sotera Defence's Distributed Graph Analytics package[48].

Stage 1 of the algorithm is implemented in Giraph. Graph data is loaded into memory, and each node is assigned a unique initial community ID. Isolated vertices halt and remove themselves from further calculation. The graph's total edge weight is calculated, as required by the modularity function (Q, Equation 3.1). At each superstep the *compute()* function receives the current vertex's neighbors' community membership and Σ values (sums of edges within and to that vertex's community). For each potential community move, change in modularity (ΔQ) is calculated. The vertex moves to the community with the highest increase in ΔQ. When no further change in modularity occurs, the vertex votes to halt.

Stage 2 of the algorithm occurs once all vertices have voted to halt the Giraph job. The Giraph task produces an updated edge list with new community membership values for each vertex. A MapReduce task compresses this graph to represent each community as a single vertex. *Map()* aggregates vertices by community ID. *Reduce()* outputs a new edge list. The reducer's current community is compressed to be represented by a single vertex, with edges internal to the community removed. Vertex-level community membership is preserved separately for later use by our method.

The compressed graph is given as input to the Stage 1 Giraph program to further improve modularity. Stages 1 and 2 are run repeatedly until no further modularity improvements occur.

Step 2: Sub-graph Layout

Our graph is now partitioned into large communities. Edge lists are retrieved for each community's graph, and stored in HDFS. Edges to vertices outside of the community will be cut.

A Python script on the cluster's master node organizes sending each community, one at a time, to a single cluster node for processing. Only the largest community subgraphs are laid out at this stage. Communities that meet this criteria are those within the top 20% by size. This value was chosen empirically. As community size follows a power-law distribution, the top 20% of communities are likely to contain the majority of the entire graph's vertices. All computing resources of each compute node are dedicated to parallel SFDP layout of its current sub-graph. The SFDP algorithm is implemented in OpenMP C++, facilitating shared-memory parallelization of vertex pair force calculations.

The output of this stage are x, y coordinates for each vertex in each processed community. These are stored in a Hive/Impala table for use in the next stage.

Step 3: Metagraph Layout

The next step begins when each significant community has been laid out. Coordinate lists for these communities, which are currently stored in HDFS, are aggregated to the master node.

The precomputed coordinates from these communities are used as the starting coordinates for a final run of SFDP layout for the entire graph. Where a node was not part of one of the communities precomputed across the cluster, it is given a random x, y starting position.

Nodes that were clustered together during the Louvain modularity step should also appear close together in a force-directed layout. We will use this idea to reduce the number of calculations needed during the final overall layout. The maximum number of iterations for pairs of precomputed vertices (tracked in a hashtable) may be capped at this stage. This maximum iteration value can be found empirically as a tradeoff between completion time and layout quality, and may be set to infinity. Global layout stability may occur before this cap is reached for graphs where the modularity partitioning is particularly well matched to the final layout positions.

During design of the method, an experimental cap of 10,000 iterations was rarely reached. Therefore, an unlimited maximum iteration value was set for our evaluations. However, further investigation of the effect of an iteration cap on completion time may be worth undertaking in future work.

Step 4: Image Generation

At this step we have 2D coordinates for every vertex in the graph, from which we can generate a vector or raster image — the final visualization of the network. `graph-tool`[46] is a Python package that allows creation of graph visualizations from existing coordinate files. It has extensive features to customize the appearance and styling of the visualization, including node size and color, and edge-bundling techniques.

The main goal of this thesis is the method to generate the layout coordinates; determining the best appearance of the visualizations is beyond the scope of this work. However, `graph-tool` provides a simple way to see basic results of our method. The Conclusion & Future Work chapter discusses possibilities to expand on this last stage.

`graph-tool` is not distributed, it runs on a single computer, but it is written in OpenMP C++ for parallel mutli-core performance. Generating images for large graphs takes negligible time compared to the process of generating the layout coordinates themselves. However, it should be noted that creating static raster images, such as JPEGs, is recommended for large graphs. While `graph-tool` provides the ability to create vector images, such as SVG and PDF formats, these are significantly slower to generate, and the resulting file may be too large to open and view.

An example of a network visualization image made with our method is in Figure 3.2. This is the Gowalla network, as described in the datasets section of the Evaluation chapter.

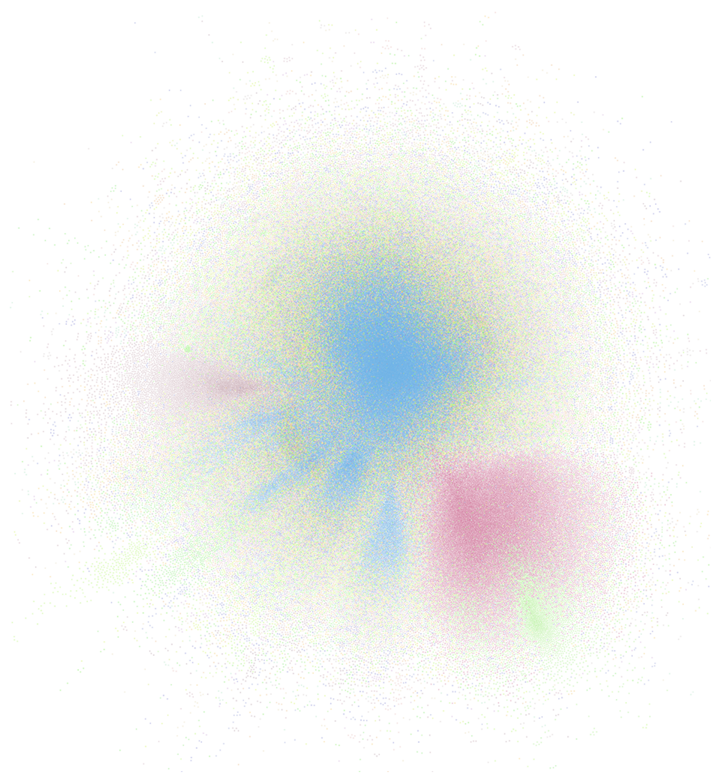

Figure 3.2: A visualization of the Gowalla network (described in the Evaluation chapter). Nodes are colored according to their community membership. We chose not to draw edges to improve clarity.

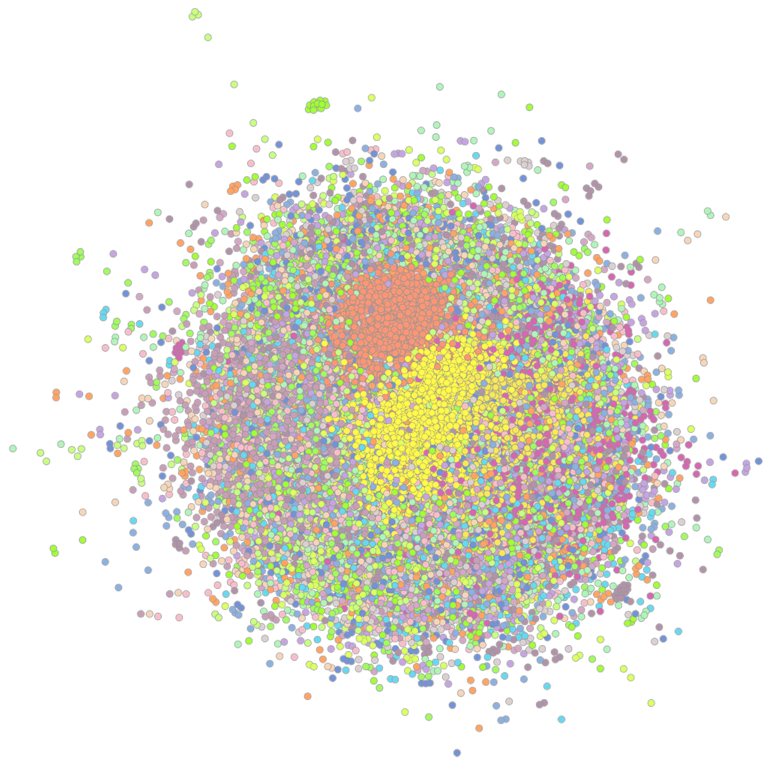

Figure 3.3: A visualization of our Facebook dataset (described in the Evaluation chapter; a different Facebook dataset than is shown in Figure 1.1). A large number of vertex pairs, isolated vertices, and smaller communities were originally in the periphery of this visualization. They were omitted due to space constraints.

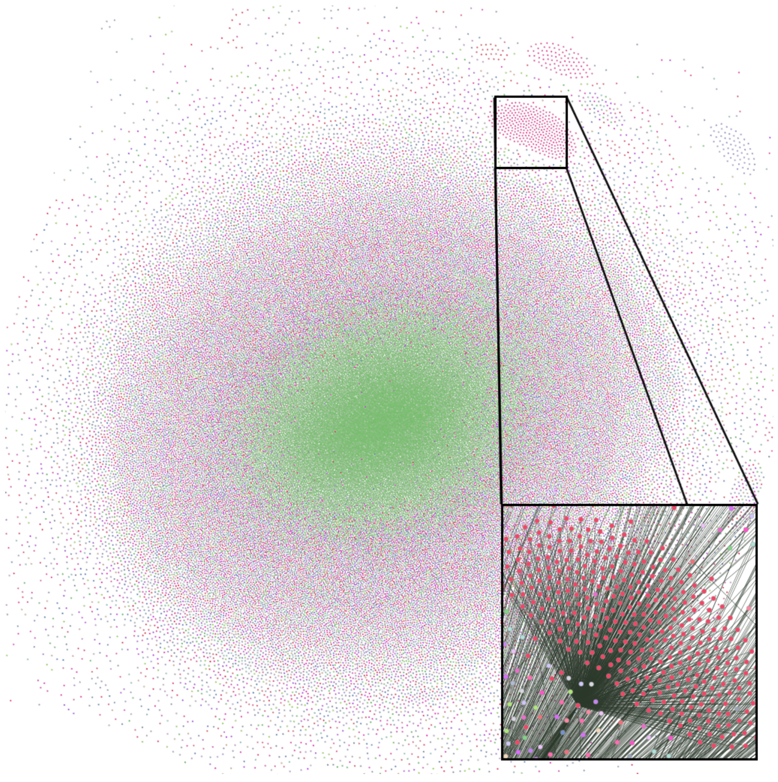

Figure 3.4: A visualization of our Slashdot test network (described in the Evaluation chapter). While we again omit drawing edges at the fully zoomed out level, we include for demonstration a zoomed in view of individual nodes and edges.

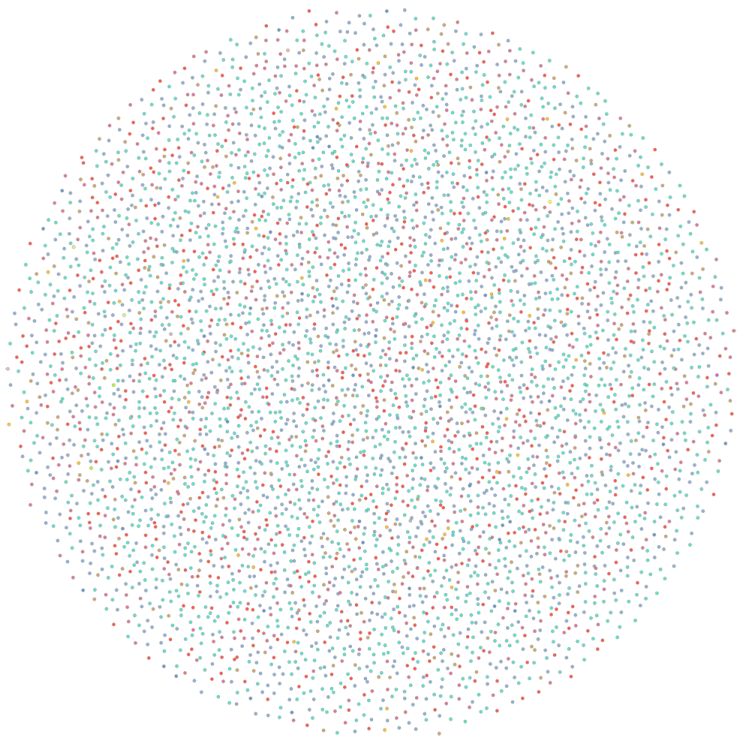

Figure 3.5: A visualization of our random Erdős-Rényi network shows that a highly symmetrical layout was produced.

Chapter 4

Evaluation

We ran several experiments to evaluate our method. This chapter first describes the data used in our testing. We worked with social network graph data that was both publicly available and previously used in other research, and two algorithmically generated datasets.

Next, we describe the methodologies used to evaluate the effectiveness of our method. In determining "effectiveness", we have two objectives to satisfy: that our method is scalable in a distributed computing environment, and that our generated layout reproduces a layout similar to the non-distributed SFDP algorithm previously evaluated in the literature.

Finally, we show the results from the tests we ran on our lab's 3-node Hadoop cluster. We discuss the results, general observations, and several findings and conclusions that can be drawn from our testing. We also discuss some of the practical and methodological limitations of our evaluation.

4.1 Data Sets

We chose to test our method primarily by laying out real-world social network graphs. Graph data from online social networks is readily available, both in repositories of previously collected data, as well as through open APIs where social graphs can be programmatically downloaded. The data we used was primarily from existing sources, and was previously used in other academic studies.

Table 4.1 lists all of the datasets used, as well as several properties important to understanding their overall structure. All of our graph datasets are from the KONECT[37] repository at the University of Koblenz–Landau, except for the Pokec network from the SNAP[39] repository at Stanford University, and our two generated datasets.

The graph properties included in Table 4.1 are: order, size, clustering coefficient

Dataset	**Source**	**Order**	**Size**	**CC (%)**	**CPL**
Facebook	KONECT	63,731	817,035	14.8	4.31
Flickr	KONECT	2,302,925	33,140,017	10.8	5.46
Pokec	SNAP	1,632,803	30,622,564	10.9	5.2
Gowalla	KONECT	196,591	950,327	2.35	4.43
Slashdot	KONECT	79,120	515,397	2.37	3.99
Erdős-Rényi	Generated	5000	6,249,491	50.0	0.5
Lattice	Generated	40,000	79600	0.0	132.33

Table 4.1: Datasets used in our evaluation, with descriptive properties; Size, Order, Clustering Coefficient (CC), and Characteristic Path Length (CPL).

(abbreviated as CC), and characteristic path length (CPL)[1]. These properties are defined as follows:

Order and *size* refer to the number of vertices and edges in the graph, respectively.

Clustering coefficient, *CC*, previously discussed in the Background chapter, is a measure of how much the nodes in a graph tend to cluster together. At the vertex level, c represents the ratio of possible edges of the vertex's neighbors, versus the number of edges that actually exist. When $c = 1$ all neighbors of that vertex have edges between them. In Table 4.1, *CC* is the average clustering coefficient for each vertex in the graph. This average can help us to gauge how closely connected the communities are in a social network. Such gauging is important to consider in relation to our work, as our method relies on community detection-based clustering to partition the workload. The clustering coefficient will affect the size and number of clusters detected, which may impact the speed and scalability of our method.

Characteristic path length, *CPL*, is a property of the entire graph for the average of the shortest path lengths between each pair of vertices. This property can be used as an indication of a small-world network. *CPL* is low in small-world networks, as shortcuts are frequently present between nodes. This is the case for all of our test social network datasets.

[1]Properties for KONECT-sourced datasets where included in the repository. We calculated the properties of the Pokec network using PEGASUS[34], a Hadoop-based analysis tool for large graphs. We calculated descriptive statistics of our generated datasets with the open source KONECT toolbox for Matlab.

4.1.1 Criteria for Inclusion

Our method is designed for social networks — specifically small-world, scale-free networks — and, in general, we expect these networks to exhibit lower clustering coefficients (often between 2–15%), and low characteristic path lengths (often 4–6). The majority of the networks in the KONECT and SNAP repositories had these characteristics. However, we excluded some specialized networks that fell greatly outside this range. For example, Choudhury et al.[13] collected a Twitter network graph by sampling the entire network's interactions with a relatively small number of other users. The result is a graph with a very low clustering coefficient of 0.06. This dataset, and others like it, would likely not be an ideal candidate for a good distributed division of work in our layout method.

Next, we narrowed our choice of graphs by their size and order. To demonstrate an effect on speedup, we wanted a collection of graphs of varying sizes. We found empirically that datasets greater than several million or tens of millions of nodes or edges took unreasonably long to layout on our cluster to test, and could be excluded. Likewise, datasets smaller than several tens of thousands of nodes or edges were not worth testing as the overhead from Hadoop's start-up phase dominated computation time, and eliminated any possible speedup.

Let us look at each dataset more closely to discuss the source of the data, and how it is appropriate for our evaluation:

Facebook: From Viswanath et al.[57], this dataset is a friendship activity network collected in 2009 from Facebook users in New Orleans. It was originally collected to understand how Facebook users interact with each other over time. This graph is among the smallest in our set, and is relatively highly clustered due to its geographic homogeneity.

Flickr: From Mislove et al.[44], this large dataset is a friendship network from Flickr, a photo hosting website with social networking features. The dataset was originally collected in 2008 over a period of three months. It was used to study the growth of online networks.

Pokec: From Takac and Zabovsky[51] in the SNAP repository, this dataset represents friendships on a popular Slovakian social network. This is the second largest dataset used in our evaluation. It was collected in 2012 to preserve the network's entire structure. The clustering coefficient and characteristic path length are similar to the slightly larger Flickr network.

Gowalla: From Cho et al.[12], this dataset is a friendship network on the now-defunct social location-sharing network Gowalla. It was collected in 2011 to study the relationship between social networks and geographic movement. It is around the same size as the Facebook dataset, but has a very low clustering coefficient.

Slashdot: From Kunegis et al.[38], this is both a "friend" and "foe" network from the online social network Slashdot, collected in 2009. Although the network was originally used to study social network metrics that contain negative edges (a "foe"), these do not have any impact within our layout method. This network exhibits a low clustering coefficient.

For the two generated networks, we used the following approaches:

Erdős-Rényi: An Erdős-Rényi graph (a random graph) was generated for use in the speedup test. We hypothesize that this network will not perform well, as it is not a small-world, scale-free graph. Our method's community detection stage should produce communities that correspond no better than random chance to a node's final layout position. The graph was generated with 50,000 nodes, with a probability of random graph edge formation of $p = 0.5$.

Lattice: A 200x200 lattice network was generated as part of our layout correctness test. A lattice should ideally layout in a perfect square formation, so any distortions caused by our algorithm's characteristics will become visually apparent. This will be explained in further detail in the Layout Correctness results.

Taken together, these datasets represent a diverse collection of real-world social networks that can realistically resemble typical use cases of our system. They span a wide range of characteristics that are important to consider in a thorough evaluation.

By using real-world datasets our results have context that is helpful for basic validation throughout the design and evaluation of our system. For example, we can quickly verify that a layout prominently shows a large community we know to exist in a network. The two remaining generated datasets help us to ensure our results are consistent with our hypotheses.

4.2 Evaluation Methods & Results

We will evaluate our method with three approaches. First, we start the evaluation with an analysis of our algorithm's complexity, to get a general assessment of the proposed method's performance. We then follow with quantitative tests: a speedup test, and a layout correctness test. This section will describe each of these tests and their results.

The evaluation methods we use each have a basis in the literature. The speedup test is the de facto evaluation method used widely in distributed and parallel computing literature, including in work based on Hadoop and Giraph. The test was employed in recent Hadoop-based work by Schumacher et al.[47] on distributed gene sequencing, and Giraph-based work by Mertella et al.[43] evaluating a general purpose partitioning algorithm, to name only a few.

Our layout correctness test is a synthesis of different approaches taken in graph layout evaluation literature. Maaten & Hinton[21] evaluate their layout method in part by comparing pairwise Euclidean distances between test datasets of different dimensionality. Bourqui et al.[9] suggest in future work correlating their layout method's Euclidean distances to graph theoretic distance. Our layout correctness test draws from both in correlating Euclidean distance between our distributed layout method and the non-distributed method upon which it is built.

4.2.1 Complexity Analysis

In a naïve, non-optimized force-directed layout, computational complexity is dominated by the calculation of forces between each pair of vertices: $O(|V|^3 + |E|)$, where $|V|$ is the graph size, and $|E|$ is its order. While our method distributes its workload across a cluster, it must also considerably reduce this cubic running time to be effective. As our method is based on the SFDP layout algorithm, we use it as the starting point of our analysis.

SFDP employs two approaches to reduce complexity: multilevel layout, and Barnes-Hut approximation. Multilevel layout collapses edges, grouping adjacent vertices, to allow sections of the graph to be moved at once initially. Barnes-Hut subdivides the graph space into a tree, allowing distant vertices to be ignored or treated as a group when calculating attraction or repulsion forces. SFDP is known to run at $O(|V|log|V| + |E|)$.

Because our approach is based off SFDP, we can expect the same complexity for the layout of individual components of the network, keeping in mind that this is performed in parallel across our cluster of size P: $O(\frac{|V|log|V|+|E|}{P})$.

We also must account for the complexity of the pre-processing stage of our method; Louvain modularity calculation that is estimated as $O(|V|log|V|)$. As Louvain is run in distributed mode, the complexity is also divided by P.

The SFDP and Louvain components are the main factors in calculating the overall complexity; however, we must also account for the complexity of the final stage — metagraph layout. The complexity of the final stage depends greatly on how well the Louvain partitioning correlates with the final layout position. We assume a coefficient to our P value, X, where $0 < X < 1$ depending on the strength of this correlation. The overall complexity of our method becomes $O(\frac{(|V|log|V|)+(|V|log|V|+|E|)}{P*X})$.

In short, despite the additional steps such as the pre-processing of Louvain modularity, due to the distributed nature of our method we can expect an improved result. As the estimated complexity makes several assumptions, we need to run an empirical test.

4.2.2 Speedup Test

The speedup test is designed to test if our method effectively utilizes the distributed computing resources made available to it. Good speedup ensures the time taken to complete a task decreases near-linearly as resources are added, while similarly scale-up ensures the size of the data possible to process increases. The methodology of the speedup test and its variables and measurements will be described.

The speedup test measures the time taken to complete a given task as several variables are controlled, including the cluster size, and the input data. A distributed task is generally considered to have good speedup if the measured time follows a linear or near-linear decrease as the cluster size increases.

However, due to many external factors, including randomness in CPU scheduling and communication latency, the time taken to compute a task can vary widely between tests. Touati et al.[54] describe a rigorous framework for a statistically valid speedup test, the overall guidelines of which we follow. Tests are run repeatedly and checked for statistical significance.

Implementation of our speedup test is structured as follows:
The **independent variables** are:

- **I1**: Number of compute nodes in the cluster (1–3)
- **I2**: Input graph (with varying properties)

The test conditions cover each combination of independent variables. For each condition we are measuring these **dependent variables**:

- **D1**: Elapsed time (wall clock time in seconds)
  - Measured with the Linux `time` function
- **D2**: Relative speedup value
  - Measured as: $time_{original}/time_n$
  - In other words: the ratio of the *original* time to run the test with 1 node to the time to run the test with n nodes.

Additionally, several other variables are recorded for informational purposes, including memory usage and any occurrences of memory paging, as these give a general indication that the program is running efficiently.

Test Environment

Our implementation and evaluation ran on a small three computer Hadoop cluster in our lab. The cluster was dedicated to the evaluation, with no other users or tasks running except for ours. Our method is almost entirely CPU-bound, and increasing cluster size benefits the processing speed primarily through additional processing capacity. RAM and disk space requirements are mostly inconsequential, as even our largest test datasets do not require more than a few gigabytes to store. Each of the three nodes consisted of identical high-end hardware, with the following features:

- **CPU**: 2× Intel Xeon E5-2690, running at 2.9GHz
  - Each with 8 hyper-threaded cores, for a total of 32 logical cores per node
- **RAM**: 64GB
- **Hard disk**: 1TB at 10k RPM
- Gigabit dedicated ethernet inter-connect

With the following software:

- **OS**: Ubuntu 14.10 LTS
- Java Runtime 8
- Cloudera Hadoop Distribution 5.4
  - Apache Hadoop 2.6
  - Apache Hive 1.1
  - Impala 2.2
- Apache Giraph 1.1.0

The speedup test was run on the test datasets at least three times per condition, and the median run time was taken. Running the test three times helps to reduce any effects of variance, while still being a small enough number of rounds to complete testing within the time constraints of this work. The breakdown of median run times (wall clock seconds) per condition (cluster nodes × datasets) is shown in Table 4.2. The median run time was tested for significant variance across nodes with a Mann-Whitney test. This test methodology follows Touati et al.'s recommendations. Median runtime is chosen instead of average as it is less sensitive to outliers.

The relative speedup value ($time_{1\_node}/time_{n\_nodes}$) for each dataset is charted in Figure 4.1. A linear speedup, shown as a dashed line, represents the perfect condition — each additional computer is used to its full potential to reduce the run time.

Speedup was attained for all datasets, to varying degrees. A Mann-Whitney test verified that median run times varied significantly across the number of cluster nodes ($0.05 < p < 0.025$). As expected, social network datasets performed much better than

Dataset	$\tilde{x}$ run time non-Dist.	$\tilde{x}$ run time 1 node	$\tilde{x}$ run time 2 nodes	$\tilde{x}$ run time 3 nodes
Facebook	585.013	907.27	630.24	408.143
Flickr	724,322	1,166,158	728,839	477,934
Pakec	577,312	929,885	598,767	366,861
Gowalla	1331.74	2182.84	1774.67	1160.64
Slashdot	656.24	1023.36	793.3	503.94
Erdős-Rényi	1361.27	3173.21	2688.98	2234.5

Table 4.2: Median ($\tilde{x}$) run times in seconds for each dataset for the non-distributed version, and the distributed version varied by nodes in the Hadoop cluster.

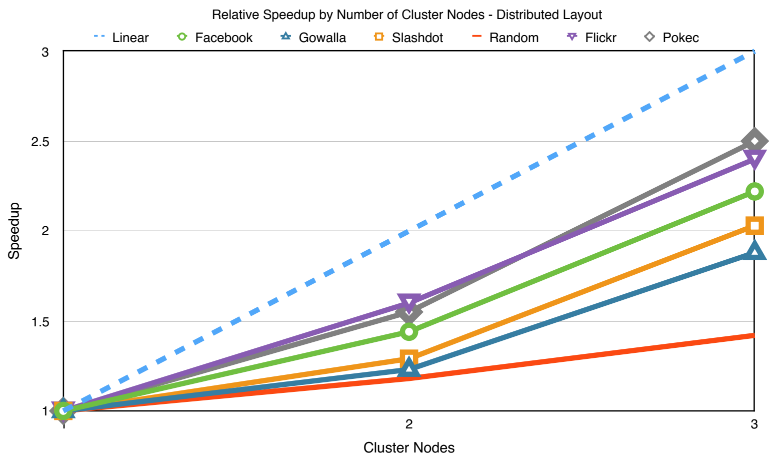

Figure 4.1: This chart shows the median relative speedup of each dataset as nodes are added to our test cluster. The dashed line represents perfect linear speedup.

the random graph. In all cases, speedup was sub-linear, with an initial indication that datasets with higher clustering coefficient seemingly performed better (although this cannot be shown with statistical significance).

The random graph performed very poorly in terms of speedup, with 3 computers performing the layout only ≈1.4x faster than 1 computer. This can be attributed to a poor graph partitioning. Community detection would not find meaningful clusters in this dataset, as the degree distribution would be close to a normal distribution. The sub-graph layout stage would not meaningfully improve the initial starting co-ordinates, and so the metagraph layout stage does not complete faster.

In many cases the standard non-distributed SFDP layout time was only matched or beaten by the distributed method after the cluster size reached 3 nodes. At first this may seem to reflect poorly upon our method's performance, but we believe this is not the case. The distributed method does add communication and coordination overhead from Hadoop and Giraph, in terms of communication and coordination; however, the largest additional processing time comes from running Louvain modularity. Community detection, often performed with Louvain modularity, is in fact a very common task to run on social network data[14][17]. For example, it is useful to color nodes graph visualizations, such as in Figure 3.2. It is plausible that users of non-distributed SFDP would run community detection on their data anyway, in which case our method would likely outperform even on a 2-node cluster. Further, we believe that a positive speedup curve would continue on a larger cluster, in which case graphs could be processed much quicker, even if the user does not find the data from the Louvain modularity step useful.

Limitations

The biggest limitation of our speedup evaluation is the relatively small size of our Hadoop cluster. We hypothesize, but cannot claim, that any speedup exists for our method beyond a 3 node cluster. Even had we had access to a larger test cluster, our evaluation is very time consuming across all conditions, and we would need to reduce the number or size of the datasets used. We believe our work shows good findings across a breadth of realistic datasets, for an initial validation of our method.

4.2.3 Layout Correctness Test

The layout correctness ensures that the visualization generated by our method is "useful". In that, we mean that the readers of a graph drawing will be able to accurately assess important aspects of its structure, and attain key information. As the non-distributed SFDP algorithm this work expands upon has already been evaluated for usefulness, we directly compare our method against it as a gold standard. Although testing for layout usefulness could be evaluated through a human study, this can be difficult, and often a layout is evaluated through quantitative means. SFDP's original paper tests a wide variety of graphs for aspects including folding, and symmetry. The author subjectively compares to existing layout methods, and finally concludes "[SFDP] is demonstrated to be ... of high quality for large graphs ..." and "gives better drawings for some difficult problems."[30]

While our method should produce layouts very similar to SFDP, we expect it to vary from the non-distributed implementation. Such variance is due to our method's assumption that modularity-based graph partitioning will correlate highly with the final vertex positions in scale-free, small-world network graphs. As such, our method's final meta-graph layout step expects a stable layout to be found more quickly, and significantly reduces the number of iterations performed. However, it is improbable that the graph partition and final layout will correlate perfectly, and a certain percentage of vertices will not move to their stable positions before the final layout stage is terminated.

Implementation of our layout correctness test is structured as follows:
The **independent variables** are:

- **I1**: Input graph (with varying properties)
- **I1**: Mode: distributed (our method) vs. non-distributed SFDP

The test conditions cover each input graph, I1, for each algorithm mode, I2, while the **dependent variable** is measured:

- **D1**: Median Euclidean distance correlation, over 10% sample of vertex pairs

We run the tests conditions three times per dataset, to account for the inherent randomness of the layout method. A higher number of test rounds may be appropriate; however, three was chosen due to the long length of time each test takes to complete. The dependent variable takes the median of these test results. Due to the large size of the graphs, vertex pairs are sampled. The tests were run on our Hadoop cluster using all three compute nodes, as the size of the cluster should not impact layout quality.

Table 4.3 indicates the Pearson correlation coefficients for each dataset between our distributed layout and the non-distributed SFDP algorithm.

Correlation for each dataset was tested with a Fisher transformation test, finding significant correlation ($p < 0.001$) between algorithm modes. Figures 4.2 and 4.3 show the correlation charts for the Facebook and Erdős-Rényi datasets, and are helpful examples in understanding this test and its results.

Dataset	Correlation Coefficient
Facebook	0.982947
Flickr	0.987111
Pokec	0.988974
Gowalla	0.993661
Slashdot	0.979822
Erdős-Rényi	0.071922

Table 4.3: Pearson correlation coefficients of pairwise Euclidean distances between distributed and non-distributed layout for each test dataset

The Facebook correlation chart in Figure 4.2 is typical to the other social network datasets. The Euclidean distance distribution is bimodal, representing nodes belonging to the denser clusters of the network, and all other nodes, as we expect from our understanding of small-world, scale-free networks. This can be seen in the chart. The lower left has many values that do not vary greatly — these are nodes that 'live' within denser clusters. All other nodes that are not as densely connected to the graph appear in the upper right of the chart. Their distances vary more widely due to the randomness of their starting layout positions.

The Erdős-Rényi correlation chart in Figure 4.3 is considerably different. While correlation exists, it is much weaker. As no distinct clusters exist in this graph — due to its approximately normal degree distribution — the layout is highly randomized and varies considerably between runs.

Finally, a small additional test was performed on the lattice network to help illustrate the variance in layout introduced by our method. As we loosely partition the graph to set initial coordinates, we effectively lock large portions of the graph's vertices into a more limited space (to reduce calculation cost). Although we see that our layout is still of high quality, this introduced minor distortions around the periphery of the visualization. These can be seen visually in the lattice drawing of Figure 4.4. SFDP itself also contributes partially to this distortion.

We show that our method produces layout results that correlate highly with the non-distributed SFDP implementation. We use a Fisher transformation to test for

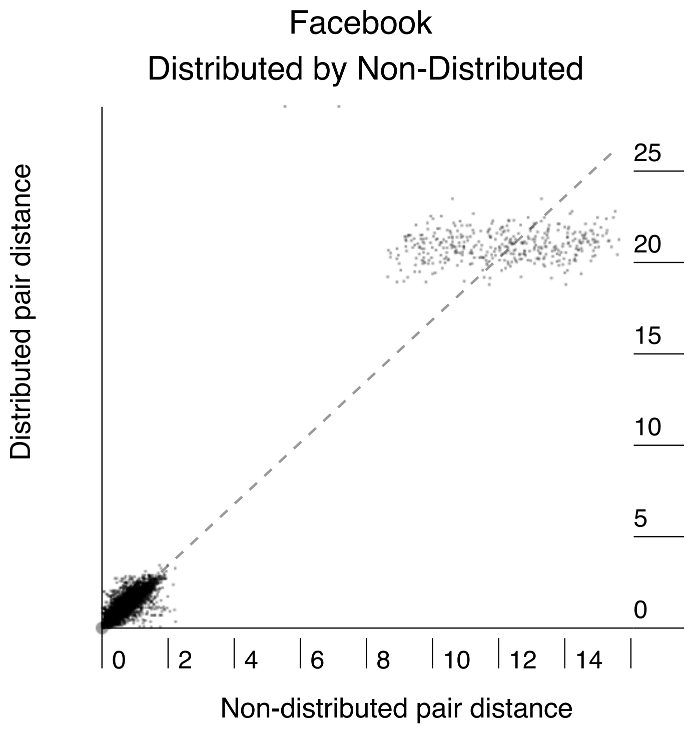

Figure 4.2: Vertex pair Euclidean distance correlation between layouts generated with distributed and non-distributed SFDP methods, for the Facebook graph. The dashed line is the correlation best fit.

correlation. Because SFDP, and consequently our method, are randomized algorithms; we must generate layouts a number of times with each method, for each of our datasets, to attain statistical power.

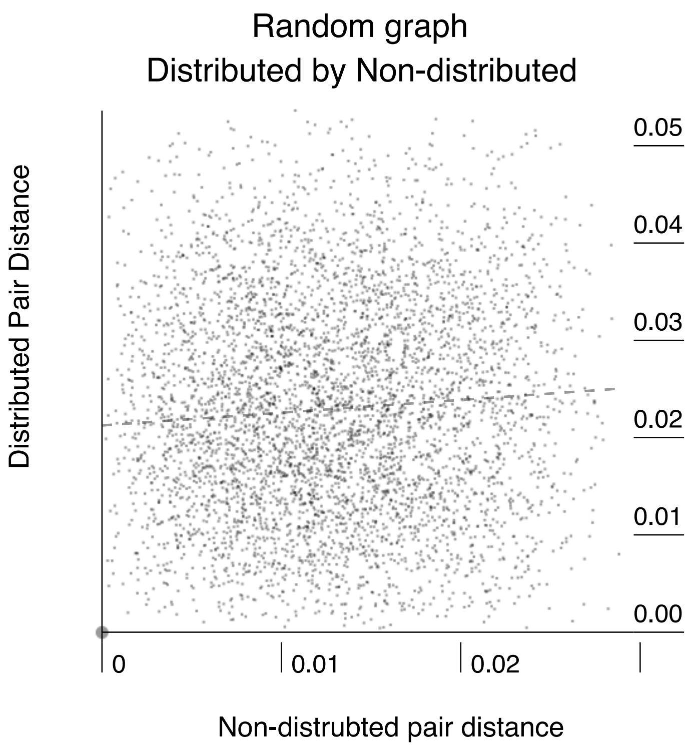

Figure 4.3: Vertex pair Euclidean distance correlation between layouts generated with distributed and non-distributed SFDP methods, for the Erdős-Rényi graph. The dashed line is the correlation best fit.

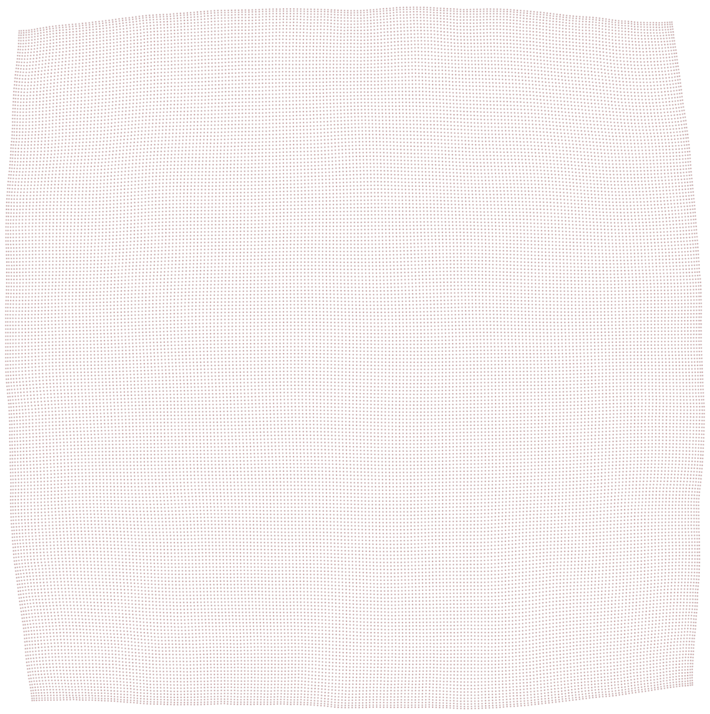

Figure 4.4: A lattice graph is laid out with our method on a 3 node cluster. Ideally the lattice will display as a perfect square in a force-directed layout. However, due to the variance between the graph's Louvain partitioning and the final layout position (exacerbated by not being a small-world, scale-free network), a distinct warping occurs. SFDP itself introduces some warping as well.

Chapter 5

Conclusion

In this work we developed a distributed computing method to generate force-directed layouts of large social network graphs. Problems in many fields can be expressed as graphs, and visual layouts of these graphs are helpful to researchers in understanding their underlying structure, in order to form hypotheses and communicate results. Notably, graph data from Online Social Networks (OSNs), now readily available to researchers, are an area of considerable research. However, these networks can easily be very large and difficult to work with. Existing layout and visualization techniques that work only on a single computer are often unable to handle data of this size in a reasonable time, if at all. Other approaches to work with large graphs may be problematic. Sampling a social network leads to the loss of its scale-free degree distribution — a key property of these networks. Dimensionality reduction based approaches to graph drawing are often limited in the precision of their layouts, possibly leading to issues in interpreting graph drawings made in this way. Therefore, a method to generate complete force-directed layouts of large social networks graphs would be a valuable contribution to the literature.

Our method takes advantage of distributed computing to allow resources to scale with the problem size. A complete force-directed layout and visualization task can be completed on larger data, in a more reasonable time, by adding computers as they are needed to a distributed cluster. To divide work between computers in the cluster, our method first partitions the graph through community detection. Our method is based on the theory that social networks follow small-world, scale-free graph characteristics: they consist of dense clusters, relatively weakly interconnected with the rest of the graph. Louvain modularity community detection provides a rough initial detection of these clusters, that should correlate highly with their final layout position in the graph drawing. By computing the layout of these community subgraphs in parallel across the cluster, we provide better than random initial starting positions for large portions of the graph. A stable global layout can now be calculated at a much lower cost.

We implemented our method with the Giraph and Hadoop frameworks. Giraph facilitates running graph processing tasks on Hadoop, a widely used ecosystem of software to manage and implement programs on distributed clusters. Louvain modularity is implemented in the Giraph and MapReduce programming paradigms. Hadoop Oozie and Impala manage distributed graph layout, and graph data storage and communication. Our work extends and makes use of the SFDP, *Scalable Force Directed Placement*, layout algorithm by Hu[30], a layout technique designed for large graphs.

We evaluated our work through two main studies; a speedup test, and a layout correctness test. The speedup test provides a critical standard in distributed computing work; that as new resources are added to the cluster our task's run time decreases near-linearly. While some communication overhead occurs, a near-linear speedup assures that our resources are effectively put to use. Secondly, as well as performing acceptably, our method must produce useful output. The SFDP algorithm, that this work extends to a distributed architecture, produces graph layouts that have been extensively evaluated for usefulness and readability. The layout correctness test correlates our method's layout, by vertex pair Euclidean distance, with the same non-distributed SFDP output.

The results from the speedup test showed sub-linear, but good speedup for each social network test dataset. Our testing used a relatively small 3-node distributed cluster; nonetheless, our work shows speedup across a breadth of realistic datasets, for a good initial validation of our method. Likewise, the layout correctness test showed a high correlation between our method's output and that of the non-distributed version.

Our method opens up many possibilities for future work beyond the scope of this thesis. These can be thought of in two categories; work to improve upon our method, and work to further evaluate it.

Technology related to distributed graph processing is improving rapidly, pushed both by social media companies and academia. Even during the development of this thesis, new technologies and improvements to existing techniques have emerged. Alternatives to Giraph and Hadoop may be evaluated for potential improvements to scalability and speedup. These include GraphX by Xin et al.[60], and Distributed GraphLab by Low et al.[40].

In addition to technology and implementation work, exploration into algorithmic improvement of our method is possible. Alternatives to Louvain modularity and SFDP, that our method uses for graph partitioning and layout respectively, can be considered.

Louvain modularity is inherently unstable compared to other community detection techniques; detected communities in a graph can vary over different runs of the algorithm. Louvain's natural tendency to form large communities results in vertices being added to communities purely due to randomness. While we justified the speed of Louvain as outweighing this weakness, other graph partitioning methods may be appropriate and should be further evaluated for their trade-offs of speed and layout quality. Fortunato[17] provides a very extensive survey of community detection and graph partitioning methods.

There is great potential for more in depth evaluation and study of our method. In particular a more controlled experimental design is possible by testing with generated datasets. While our work focused on showing an initial and practical demonstration of our method with real-world datasets, algorithmically generated data can be more precisely manipulated. Detailed effects of clustering coefficient and other graph properties on layout quality and speedup could be revealed.

Our available computing resources limited our speedup study to a three node cluster. Further evaluation on larger Hadoop clusters would reveal how our method's speedup behaves as communication and coordination needs increase. A larger cluster would allow larger datasets to be laid out and evaluated than we were able to test, due to time limitations.

Finally, we propose that this thesis work could be applied to tasks beyond network layout and visualization. This work can be thought of as a starting framework for other scale-free distributed graph processing tasks. Other divide and conquer-like algorithms, where large subgraphs can be manipulated independently in parallel and finally recombined, may be ideal candidates.

Bibliography

[1] Nathanael L. Ackerman, Cameron E. Freer, and Daniel M. Roy. On the computability of conditional probability. *Advances in Neural Information Processing Systems*, page 30, 2010.

[2] Alex T. Adai, Shailesh V. Date, Shannon Wieland, and Edward M. Marcotte. LGL: Creating a map of protein function with an algorithm for visualizing very large biological networks. *Journal of Molecular Biology*, 340(1):179–190, 2004.

[3] Alexa. *Alexa Top 500 Global Sites*. www.alexa.com/topsites, 2015.

[4] Luís A. Nunes Amaral, Antonio Scala, Marc Barthelemy, and H E Stanley. Classes of small-world networks. *Proceedings of the National Academy of Sciences of the United States of America*, 97(21):11149–11152, 2000.

[5] Reza Bakhshandeh and M Samadi. Degrees of separation in social networks. *Fourth Annual Symposium on Combinatorial Search*, pages 18–23, 2011.

[6] Josh Barnes and Piet Hut. A hierarchical O (N log N) force-calculation algorithm. *Nature*, 324(4):446–449, 1986.

[7] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An Open Source Software for Exploring and Manipulating Networks. In *Third International AAAI Conference on Weblogs and Social Media*, pages 361–362, 2009.

[8] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 10008(10):6, 2008.

[9] Romain Bourqui, David Auber, and Patrick Mary. How to draw clustered weighted graphs using a multilevel force-directed graph drawing algorithm. *Proceedings of the International Conference on Information Visualisation*, pages 757–764, 2007.

[10] Andreas Buja, Deborah F Swayne, Michael L Littman, Nathaniel Dean, Heike Hofmann, and Lisha Chen. Data Visualization with Multidimensional Scaling. *Journal of Computational and Graphical Statistics*, 06511:1–30, 2007.

[11] Avery Ching and C Kunz. Giraph: Large-scale graph processing infrastructure on Hadoop. *Hadoop Summit*, 6(29):2011, 2011.

[12] Eunjoon Cho, Seth A. Myers, and Jure Leskovec. Friendship and mobility. *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '11*, page 1082, 2011.

[13] Munmun De Choudhury, Yu-Ru Lin, Hari Sundaram, K. Selcuk Candan, Lexing Xie, and Aisling Kelliher. How does the data sampling strategy impact the discovery of information diffusion in social media? *Proceedings of the 4th International AAAI Conference on Weblogs and Social Media*, pages 34–41, 2010.

[14] David Combe, Christine Largeron, Elod Egyed-Zsigmond, and Mathias Géry. A comparative study of social network analysis tools. *International Workshop on Web Intelligence and Virtual Enterprises*, 2(2010):1–12, 2010.

[15] Jeffrey Dean and Sanjay Ghemawat. MapReduce. *Communications of the ACM*, 51(1):107, 2008.

[16] Marion Dierickx and Stephen Portillo. *N-Body Building.* www.portillo.ca/nbody, 2013.

[17] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, 2010.

[18] Linton C Freeman. Visualizing Social Networks. *Journal of Social Structure*, 1(1), 2000.

[19] Linton C Freeman, Dorothea Wagner, and Ulrik Brandes. Social Networks. In *Handbook of Graph Drawing and Visualization*, pages 805–839. 2013.

[20] Thomas M. J. Fruchterman and Edward M. Reingold. Graph Drawing by Force-directed Placement. *Software-Practice and Experience*, 21:1129–1164, 1991.

[21] Carlos R. García-Alonso, Leonor M. Pérez-Naranjo, and Juan C. Fernández-Caballero. Multiobjective evolutionary algorithms to identify highly autocorrelated areas: The case of spatial distribution in financially compromised farms. *Annals of Operations Research*, 219:187–202, 2014.

[22] Alexandros V. Gerbessiotis and Leslie G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, 1994.

[23] Helen Gibson, Joe Faith, and Paul Vickers. A survey of two-dimensional graph layout techniques for information visualisation. *Information Visualization*, 12(3-4):324–357, 2012.

[24] Barbara A. Given and Marcia Grant. Introduction. In *Seminars in Oncology Nursing*, volume 27, pages 91–92. 2011.

[25] Yong Guo, Ana Lucia Varbanescu, Alexandru Iosup, Claudio Martella, and Theodore L. Willke. Benchmarking graph-processing platforms: A vision. *Proceedings of the 5th ACM/SPEC international conference on Performance engineering - ICPE '14*, pages 289–292, 2014.

[26] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. The rise of Big Data on cloud computing: Review and open research issues. *Information Systems*, 47:98–115, 2014.

[27] Nathalie Henry, Jean-Daniel Fekete, and Michael J McGuffin. NodeTrix: a Hybrid Visualization of Social Networks. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1302–1309, 2007.

[28] Michael A Hogg, Joel Cooper, Phoebe C Ellsworth, Richard Gonzalez, Steven J Sherman, Matthew T Crawford, David L Hamilton, and Leonel Garcia-marques. *The SAGE Handbook of Social Network Analysis*, volume 88. SAGE publications, 2011.

[29] Danny Holten and Jarke J. van Wijk. Force-Directed Edge Bundling for Graph Visualization. *Computer Graphics Forum*, 28(3):983–990, 2009.

[30] Yifan Hu. Efficient and High Quality Force-Directed Graph Drawing. *The Mathematica Journal*, 10(1):37–71, 2005.

[31] Mathieu Jacomy, Tommaso Venturini, Sebastien Heymann, and Mathieu Bastian. ForceAtlas2, a continuous graph layout algorithm for handy network visualization designed for the Gephi software. *PLoS ONE*, 9(6):1–12, 2014.

[32] Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, Yun Tian, James Majors, Adam Manzanares, and Xiao Qin. Improving MapReduce performance through data placement in heterogeneous Hadoop clusters. *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–9, 2010.

[33] Tomasz Kajdanowicz, Wojciech Indyk, Przemyslaw Kazienko, and Jakub Kukul. Comparison of the efficiency of MapReduce and Bulk Synchronous Parallel approaches to large network processing. *Proceedings of the 12th IEEE International Conference on Data Mining Workshops, ICDMW 2012*, pages 218–225, dec 2012.

[34] U Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: mining peta-scale graphs. *Knowledge and Information Systems*, 27(2):303–325, 2011.

[35] V. D. Kukin. Delta-function. *Encyclopedia of Mathematics*, 2011.

[36] Ravi Kumar, Jasmine Novak, and Andrew Tomkins. Structure and evolution of online social networks. *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, page 611, 2006.

[37] Jérôme Kunegis. The Koblenz Network Collection. In *Proceedings of the 22Nd International Conference on World Wide Web Companion*, pages 1343–1350, 2013.

[38] Jérôme Kunegis, Andreas Lommatzsch, and Christian Bauckhage. The slashdot zoo. *Proceedings of the 18th international conference on World wide web - WWW '09*, page 741, 2009.

[39] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data, 2014.

[40] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning in the Cloud. pages 716–727, 2012.

[41] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[42] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–145, 2010.

[43] Claudio Martella, Dionysios Logothetis, and Georgos Siganos. Spinner: Scalable Graph Partitioning for the Cloud. *arXiv preprint arXiv:1404.3861*, pages 1–15, 2014.

[44] Alan Mislove, Hema Swetha Koppula, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Growth of the flickr social network. *Proceedings of the first workshop on Online social networks - WOSP '08*, pages 25–30, 2008.

[45] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and Analysis of Online Social Networks. *Imc*, 2007.

[46] Tiago P. Peixoto. The graph-tool python library. *Figshare*, 2014.

[47] André Schumacher, Luca Pireddu, Matti Niemenmaa, Aleksi Kallio, Eija Korpelainen, Gianluigi Zanetti, and Keijo Heljanko. SeqPig: Simple and scalable scripting for large sequencing data sets in hadoop. *Bioinformatics*, 30(1):119–120, 2014.

[48] Sotera Defence. Distributed Graph Analytics, 2014.

[49] Michael P H Stumpf, Carsten Wiuf, and Robert M May. Subnets of scale-free networks are not scale-free: sampling properties of networks. *Proceedings of the National Academy of Sciences of the United States of America*, 102(12):4221–4, 2005.

[50] Kozo Sugiyama. *Graph Drawing and Applications for Software and Knowledge Engineers*. World Scientific, 2002.

[51] Lubos Takac and M Zabovsky. Data Analysis in Public Social Networks. *International Scientific Conference & International Workshop*, (May):1–6, 2012.

[52] Roberto Tamassia. Handbook of Graph Drawing and Visualization. *CRC Press*, page vii, 2013.

[53] Anna Tikhonova and Kwan-liu Ma. A Scalable Parallel Force-Directed Graph Layout Algorithm. *Eurographics Symposium on Parrallel Graphics and Visualization*, pages 25–32, 2008.

[54] Sid Ahmed Ali Touati, Julien Worms, and Sébastien Briais. The Speedup-Test: A statistical methodology for programme speedup analysis and computation. *Concurrency Computation Practice and Experience*, 25(10):1410–1426, 2013.

[55] Jeffrey Travers and Stanley Milgram. An Experimental Study of the Small World Problem. *Sociometry*, 32(44):425–443, 1969.

[56] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The Anatomy of the Facebook Social Graph. page 17, 2011.

[57] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. On the Evolution of User Interaction in Facebook. *Proceedings of the 2nd ACM workshop on Online social networks - WOSN '09*, page 37, 2009.

[58] Duncan J. Watts. The New Science of Networks. *Annual Review of Sociology*, 30(1):243–270, 2004.

[59] Duncan J. Watts and Steven Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–2, 1998.

[60] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. GraphX. *First International Workshop on Graph Data Management Experiences and Systems - GRADES '13*, pages 1–6, 2013.

[61] E Yunis, R Yokota, and A Ahmadia. Scalable Force Directed Graph Layout Algorithms Using Fast Multipole Methods. *Parallel and Distributed Computing (ISPDC), 2012 11th International Symposium on*, pages 180–187, 2012.