A Comparison of Techniques for Graph Analytics on Big Data

by

Muhammad Usman Nisar

(Under the Direction of John A. Miller)

Abstract

Graphs enjoy profound importance because of their versatility and expressivity. They can be effectively used to represent social networks, search engines and genome sequencing. The field of subgraph pattern matching has been of significant importance and has wide-spread applications. Conceptually, we want to find subgraphs that match a pattern in a given graph. Much work has been done in this field with solutions like Subgraph Isomorphism and Regular Expression matching. With Big Data, scientists are frequently running into massive graphs that have amplified the challenge that this area poses. We study the speedup and communication behavior of three distributed algorithms that we proposed for inexact pattern matching. We also study the impact of different graph partitionings on runtime and communication. Our extensive results show that the algorithms exhibit excellent scalable behavior and min-cut partitioning can lead to improved performance under some circumstances, and can also drastically reduce the network traffic.

Index words: pattern matching, simulation, graph partitioning, parallel and distributed algorithms, vertex-centric

A Comparison of Techniques for Graph Analytics on Big Data

by

Muhammad Usman Nisar

B.S., National University of Engineering and Computer Sciences, Islamabad, Pakistan, 2006

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2013

A Comparison of Techniques for Graph Analytics on Big Data

by

Muhammad Usman Nisar

Approved:

Major Professor: John A. Miller

Committee: Lakshmish Ramaswamy
Ismail B. Arpinar

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2013

A Comparison of Techniques for Graph Analytics on Big Data

Muhammad Usman Nisar

April 25, 2013

Dedication

To two people that mean the most to me - **Dad** and ***Baji***, this one is for you! Thank you for the love, guidance and prayers for me every step of the way.

Acknowledgments

I would like to thank Dr. John A. Miller for his continued support, relentless help and direction throughout my degree and thesis - Thank you for everything. I would also like to thank Dr. L. Ramaswamy and Dr. Ismail B. Arpinar for their help and guidance whenever needed. I would also like to thank Arash J. Fard for being an excellent research partner and for all the fun time we had working on this project.

So many names come to my mind when I think of people who have contributed in so many ways that it would be hard to put their names here. Keeping it short, I would like to **thank all** of my friends for being there for me always.

Last but definitely not least, I would like to thank my whole family for always standing behind me. All of you guys rock - take a bow!

Contents

1 Introduction **1**

2 Background **3**

2.1 Subgraph Pattern Matching 4

2.2 Types of Pattern Matching 4

3 Models of Computation **12**

3.1 MPI-like 12

3.2 MapReduce 12

3.3 BSP - Bulk Synchronous Parallel 13

3.4 Vertex-Centric Graph Processing 15

4 Distributed Algorithms for Graph, Dual and Strict Simulation **17**

4.1 Graph Simulation 17

4.2 Dual Simulation 19

4.3 Strict Simulation 20

5 Implementation of Distributed Algorithms **23**

5.1 GPS - Graph Processing System 23

5.2 Akka 24

5.3 GPS vs Akka 25

6 Performance Evaluation **27**

6.1 Experimental Setup 27

6.2 Experimental Results 29

7 Impact of Graph Partitioning **36**

7.1 Experimental Results 40

8 Related Work **51**

9 Conclusion **53**

9.1 Future Work 54

A Results of experiments on different partitioning schemes **56**

A.1 Runtime 56

A.2 Network I/O 57

List of Figures

2.1 Subgraph Isomorphism 5

2.2 An example for Graph Simulation 7

2.3 An example for Dual Simulation 8

2.4 An example for Strong Simulation 9

2.5 A comparison of strong vs strict simulation 11

3.1 An example for Bulk Synchronous Parallel (BSP) 14

4.1 Summary of Graph Simulation algorithm 18

4.2 An example for distributed graph simulation 19

4.3 A ball around a vertex with $d_q = 2$ 20

4.4 A breadth-first ball around vertex X with $d_q = 2$ 22

5.1 Comparison between GPS and Akka using Graph Simulation 26

6.1 The workflow of the auto comparison model 29

6.2 Running times of graph, dual and strict simulation, $|V_q| = 25, \alpha_q = 1.2$ . . . 30

6.3 The speedup on the synthesized dataset, $|V| = 10^8$, $|V_q| = 25, \alpha = \alpha_q = 1.2$. 31

6.4 The speedup on the uk-2005 dataset, $|V| = 3.9\text{x}10^7$, $|V_q| = 25, \alpha_q = 1.2$ . . . 32

6.5 The speedup on the enwiki-2013 dataset, $|V| = 4.2\text{x}10^6$, $|V_q| = 25, \alpha_q = 1.2$. 32

6.6 Efficiency for the synthesized dataset, $|V| = 10^8, |V_q| = 25, \alpha = \alpha_q = 1.2$ . . . 34

6.7 Efficiency for the uk-2005 dataset, $|V| = 3.7\text{x}10^7$, $|V_q| = 25, \alpha_q = 1.2$ 34

6.8 Efficiency for the enwiki-2013 dataset, $|V| = 4.2\text{x}10^6$, $|V_q| = 25, \alpha_q = 1.2$ . . . 35

7.1 Graph partitioning 38

7.2 Partitioning effect on the runtime of synthesized dataset, $|V| = 10^7, \alpha = 1.2$. 41

7.3 Partitioning effect on the runtime of uk-2002-hc, $|V| = 1.8\text{x}10^7$ 42

7.4 Partitioning effect on the network I/O of synthesized dataset, $|V| = 10^7, \alpha = 1.2$ 44

7.5 Partitioning effect on the network I/O of uk-2002-hc, $|V| = 1.8\text{x}10^7$ 45

7.6 Partitioning effect on the runtime of synthesized dataset, $|V| = 10^7$, $\alpha_q = 1.75$ 47

7.7 Partitioning effect on the runtime of uk-2002-hc, $|V| = 1.8\text{x}10^7$, $\alpha_q = 1.75$ . . 48

A.1 Comparison of partitioning effects on the runtime of synthesized dataset . . 57

A.2 Comparison of partitioning effects on the runtime of dataset uk-2002-hc . . . 58

A.3 A comparison of partitioning effects on the network I/O of synthesized dataset 59

A.4 A comparison of partitioning effects on the network I/O of uk-2002-hc 60

List of Tables

7.1 Runtimes of strict simulation on 10 different queries, $|V_q| = 100, \alpha_q = 1.75$ . . 50

7.2 Runtimes of strict simulation on 10 different queries, $|V_q| = 100, \alpha_q = 1.2$ . . 50

Chapter 1

Introduction

Graphs are of utmost importance in the field of Computer Science because of their expressivity and the ability to abstract a huge class of problems. They have been successfully used to study and model numerous problems in different fields. These applications vary from software plagiarism detection, web search engines, study of molecular bonds to the modeling of social networks like Facebook, LinkedIn and Twitter [1].

Graph pattern matching is one of the most important and widely studied class of problems in graphs. A considerable amount of research has been put into this area, sprouting concepts like Subgraph Isomorphism, Regular Expression matching [2] and Graph Simulation [3]. Conceptually, pattern matching algorithms seek to find subgraphs of a given graph that are similar to the given pattern graph [4]. Though, the Subgraph Isomorphism returns the strictest matches for graph matching in terms of topology [5], the problem is NP-complete [6], and thus does not scale well even for medium-scale graphs.

Graph simulation, on the other hand, provides a practical alternative to subgraph isomorphism by relaxing the stringent matching conditions of subgraph isomorphism, and allowing matches to be found in polynomial time. Some researchers [4, 7, 8] even argue that graph simulation is more appropriate than subgraph isomorphism for modern problems like social

network analysis because it yields matches that are conceptually more meaningful.

With the rapid advent of Big Data, graphs have transformed into huge sizes and are rapidly getting out of the grasp of conventional computational approaches. Nowadays, graphs with millions of vertices and billions of edges are becoming a norm. New computational models and modern techniques are needed to scale to this ever-growing need of processing power. This paper discusses a few of those solutions for the subgraph pattern matching problem.

The outline of this report is as follows: In next chapter, we discuss the background and a description of the subgraph pattern matching problem along with its types. In Chapter 3, we discuss a few computational models that have been used recently for graph processing in distributed systems. We follow-up with brief description of three new algorithms that we proposed in [4]. Chapter 5 goes through the implementation of distributed algorithms on two platforms and briefly compare their pros and cons. Chapter 6 discusses the experimental results and report the speedup and efficiency of the algorithms. Chapter 7 talks about the impact of graph partitioning along two lines of runtime improvement and network I/O reduction. The penultimate chapter gives an insight into the related work and the final chapter concludes with a listing of future work.

Chapter 2

Background

In this section, we discuss the background and give motivation for the need of new approaches to deal with the large scale graphs in general and query processing in particular. Today, Facebook has over 1 billion vertices and the average degree of each vertex is 140[1], Twitter has well over 200 million active users creating over 400 million tweets each day[2]. In genome sequencing, recent work [9] attempts to solve the genome assembly problem by traversing the de Brujin graph of the read sequence. The de Brujin graph can contain as many as 4^k vertices where k is atleast 20. All of these models, translate to massive graphs that many existing approaches fail to cope with.

To handle the mammoth scale of these graphs, an obvious approach is to distribute the graphs onto multiple machines and then run them concurrently to efficiently calculate the result in parallel. Consequently, some of the basic challenges are the following:

1. how to distribute the graph?

2. how to come up with an *efficient* algorithm that runs as concurrently as possible?

3. how to reduce the communication/traffic among different machines (in part, a side

[1]http://www.facebook.com/press/info.php?statistics
[2]http://blog.twitter.com

affect of the top two)?

We go through these questions as we discuss the problem of Subgraph Pattern Matching in depth.

2.1 Subgraph Pattern Matching

The problem of subgraph matching is defined as follows: Let G=*(V, E, l)* be a graph, where V is the set of vertices, E is the set of edges, and l is the labelling function that assigns a label to each vertex in V. Let Q=*(V_q, E_q, l_q)* be the query (pattern) graph where V_q is the set of vertices, E_q is the set of edges and l_q are the labels of V_q. Intuitively, the goal of subgraph pattern matching is to find all subgraphs from the data graph G that match the pattern graph Q. Thus, $G'(V', E', l')$ is a subgraph of G if and only if (1) $V' \subseteq V$; (2) $E' \subseteq E$; and (3) $\forall u \in V' : l'(u) = l(u)$.

In this paper, without loss of generality we assume all vertices are labeled, all edges are directed, and there are no multiple edges. We also assume a query graph is a connected graph because the result of pattern matching for a disconnected query graph is equal to the union of the results for its connected components. We use the terms pattern and query graph interchangeably.

2.2 Types of Pattern Matching

In this section, we briefly review the five types of pattern matching. The first one is Subgraph Isomorphism, the next one is Graph Simulation proposed in [3], the Dual Simulation and Strong Simulation were proposed in the [10] and Strict Simulation by us [4]. A much more exhaustive list can be found at [11].

We use an example for every type to illustrate the concept. The example uses the idea

of a recommendation network where we are looking for specific expertise satisfying some conditions in a social network. All of the examples have a query graph and a datagraph with labels inside the nodes and vertex ids hanging outside the nodes.

2.2.1 Subgraph Isomorphism

Arguably, subgraph isomorphism is the most widely studied problem for graph pattern matching. By definition, subgraph isomorphism describes a bijective mapping between a query graph $Q(V_q, E_q, l_q)$ and a subgraph of a data graph $G(V, E, l)$, denoted by $Q \trianglelefteq_{iso} G$. That is, assuming $G'(V', E', l')$ is a subgraph of G, graph Q will be subgraph isomorphic of G if there is a bijective function f from the vertices of Q to the vertices of G' such that (u, v) is an edge in Q if and only if $(f(u), f(v))$ is an edge in G' [12]. It should be noted that function f ensures that u and $f(u)$ have the same labels. Ullmann's algorithm is widely known, and still is probably the most popular subgraph isomorphism algorithm [11]. This is a form of *exact* matching. However, this problem is NP-hard in general case and not scalable in case of large graphs.

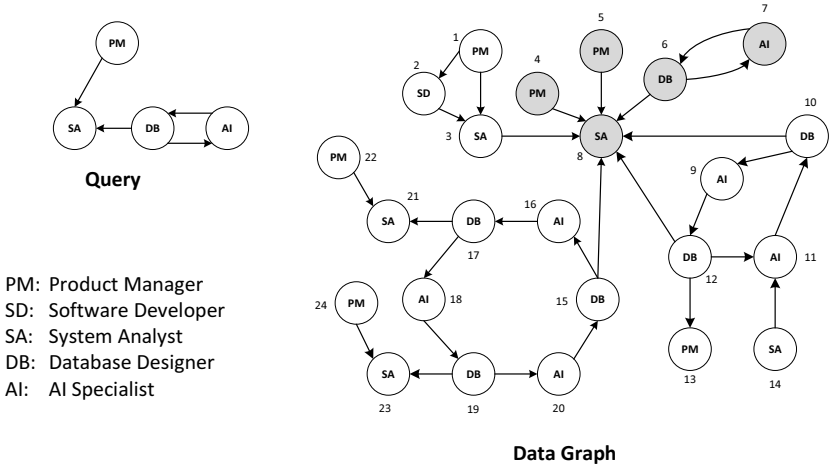


Figure 2.1: Subgraph Isomorphism

Figure 2.1 shows an example of subgraph isomorphism. There will be two subgraph matches {4,6,7,8} and {5,6,7,8} using subgraph isomorphism that are highlighted in the data graph.

2.2.2 Graph Simulation

Graph simulation allows a faster alternative to subgraph isomorphism by relaxing some conditions.

Definition: Pattern $Q(V_q, E_q, l_q)$ matches data graph $G(V, E, l)$ via *graph simulation*, denoted by $Q \trianglelefteq_{sim} G$, if there is a binary relation $R \subseteq V_q \times V$ such that (1) if $(u, u') \in R$, then u and u' have the same label; (2) for every $u \in V_q$ there is a $u' \in V$ such that $(u, u') \in R$; (3) for every $(u, v) \in E_q$ there is a $(u', v') \in E$ such that $(u, u') \in R$ and $(v, v') \in R$ [10].

Intuitively, graph simulation only captures the child relationships of vertices. HHK - a quadratic algorithm, was first proposed in [3] and efficiently computes the match set on medium-sized graphs, but fails to scale on large graphs.

In plain words, a vertex in a data graph becomes a graph simulation match with a vertex in query graph if and only if

1. both have the same label, and

2. a subset of its children match all the children of its corresponding vertex in the query graph.

If both the conditions are true for atleast a single vertex in the data graph for every vertex in the query graph, then we can say that Q is a graph simulation match to G denoted by $Q \trianglelefteq_{sim} G$.

In example 2.2, we show graph simulation in action. As can be seen, there are a lot more semantic matches than just the subgraph isomorphism. Other than the vertices {2, 13}, all the vertices are in the match set. One may argue that the result set is too big and a bit

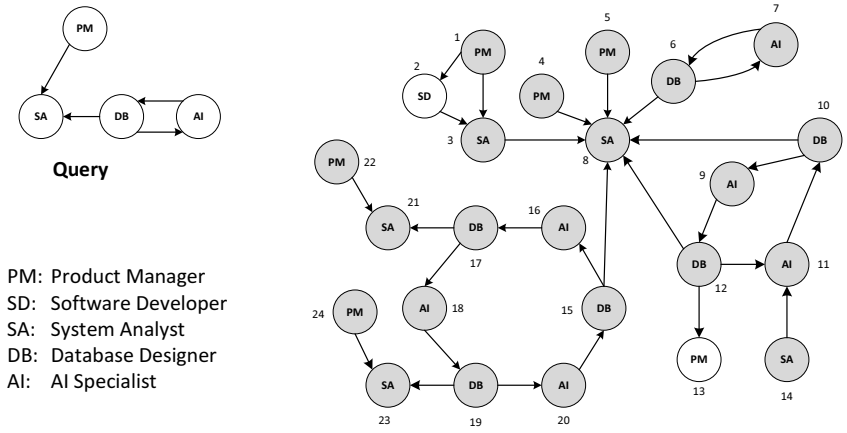


Figure 2.2: An example for Graph Simulation

irrelevant as well, e.g., {14} is a match when it is not endorsed by any PM. We will answer this question as we go through other types of similar methods below.

2.2.3 Dual Simulation

Dual simulation extends graph simulation by also taking into account the parent relationships of the vertices, thus resulting in a stricter match set.

Definition: Pattern $Q(V_q, E_q, l_q)$ matches data graph $G(V, E, l)$ via *dual simulation*, denoted by $Q \trianglelefteq^D_{sim} G$, if (1) Q is a graph simulation match to G with a match relation $R_D \subseteq V_q \times V$, and (2) for every $(u, u') \in R_D$, if there is a $w \in V_q$ such that $(w, u) \in E_q$ then there exists a $w' \in V$ such that $(w, w') \in R_D$ and $(w', u') \in E$. (adapted from [10])

That is, a vertex in a data graph becomes a dual match with a vertex in a query graph if and only if

1. both have the same label, and

2. a subset of its children match all the children of its corresponding vertex in the query graph, and

3. a subset of its parents match all the parents of its corresponding vertex in the query graph.

If all of the three conditions are true for atleast a single vertex in the data graph for every vertex in the query graph, then we can say that Q is a dual simulation match to G.

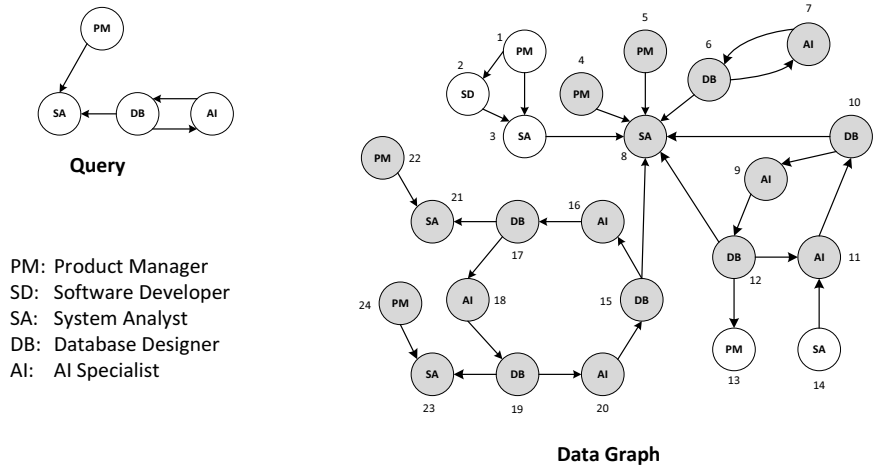


Figure 2.3: An example for Dual Simulation

Now take a look at example 2.3. We can clearly see that the vertices {1, 3, 14} have been taken out since they do not satisfy the parent condition from the pattern graph, thus resulting in a stricter result set than graph simulation.

2.2.4 Strong Simulation

Strong simulation builds on dual simulation by introducing a locality condition. The term *ball* is coined [10] to capture this aspect of the match. The ball for a vertex v with radius r contains all the vertices V_B that are within an undirected distance of r from the vertex v, moreover it has all the edges between those vertices V_B and no more.

Definition: Pattern $Q(V_q, E_q, l_q)$ matches data graph $G(V, E, l)$ via *strong simulation*, denoted by $Q \trianglelefteq^S_{sim} G$, if there exists a vertex $v \in V$ such that (1) $Q \trianglelefteq^D_{sim} \hat{G}[v, d_Q]$ with maximum dual match set R^b_D in ball b where d_Q is the diameter of Q, and (2) v is member of at least one of the pairs in R^b_D. The connected part of the result match graph of each ball with respect to its R^b_D which contains v is called a *maximum perfect subgraph* of G with respect to Q. [4]

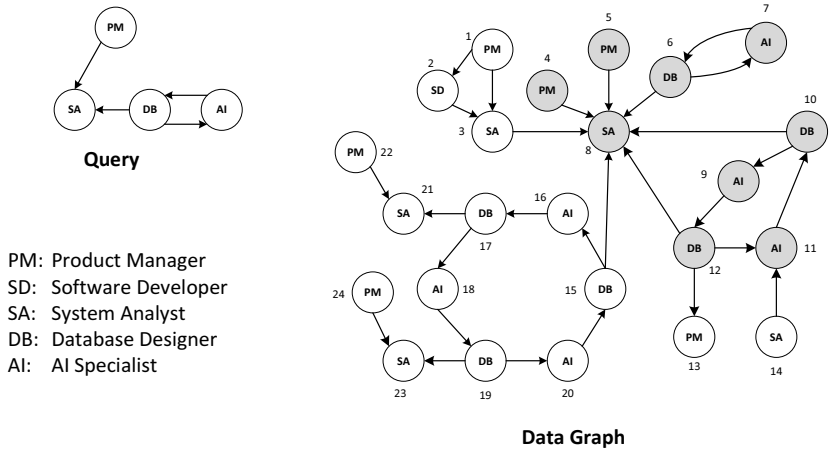


Figure 2.4: An example for Strong Simulation

We can see in Fig 2.4 that the big loop spanning nodes {15, 16, 17, 18, 19, 20, 21, 22, 23, 24} have been taken out since they do not satisfy the locality condition for any ball in the graph. As claimed earlier, this results in an even more strict match set that retains some semantic matches but ignores a few that it deems are too spread out.

2.2.5 Strict Simulation

Strict simulation, proposed in [4], is an extension and improvement over strong simulation. The fundamental difference between strong and strict simulation is the way the balls are created. Whereas in strong simulation, the ball is created from the entire data graph, in

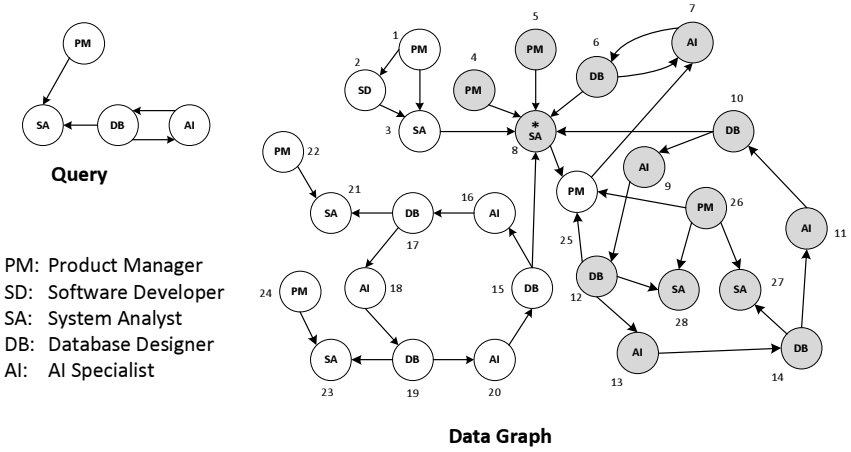
strict simulation we only create balls out of the vertices that were a dual match - thus we not only reduce much of the effort in creating the balls but also more importantly, it results in better results thar are closer to those of subgraph isomorphism [4]

We explain strict simulation in more detail in the next section by comparing it with strong simulation.

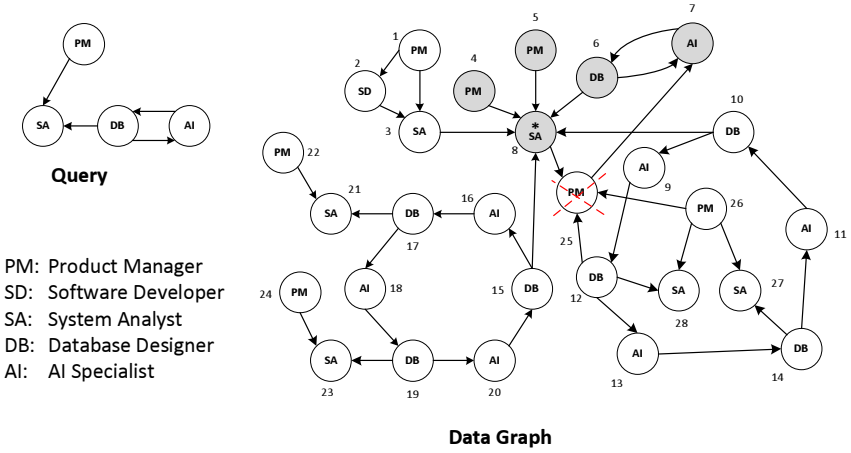
2.2.6 Comparison of Strong and Strict Simulation

We refer Fig 2.5 for the example. In the case of strong simulation, we see that because of the ball around {8}, the resultant match set to the query is quite big. We have marked {25} in white, since it is not a dual match itself but because strong simulation does not make any differentiation, the resultant ball around it will have all the vertices (marked in grey) as the resulting match set to the original query.

On the other hand in strict simulation, when we do the duality check the node {25} is taken out. Thus, no matter what ball we create, none will have anymore matches than the ones marked in grey. Also, it can be easily observed that the strict result set {4, 5, 6, 7, 8} is exactly the same as subgraph isomorphism for this example. Thus, going through all the examples, we can see how we gradually move towards matches that are more closer to subgraph isomorphism. However, it should be noted that they are not equal. For example, if there was an additional loop of {DB, AI} at {7} connecting to {6}, then strict result would have been the same as strong simulation.



(a) Strong Simulation



(b) Strict Simulation

Figure 2.5: A comparison of strong vs strict simulation

Chapter 3

Models of Computation

With the advent of Big Data computing, computational models for graph algorithms have been re-examined. Over the years, a number of ideas have been proposed for efficient and scalable processing of graphs. Since, this paper is focussed on big graphs, we only go through some of the most important distributed models.

3.1 MPI-like

Several libraries are developed using MPI during last decade to provide platforms for distributed graph processing like Parallel BGL [13] and CGMgraph [14]. However, these libraries do not support fault tolerance or some other issues that are important for very large scale distributed systems.

3.2 MapReduce

MapReduce is a programming model proposed for the processing of large data sets [15] by Google. It has been successfully used and deployed worldwide for the parallel computation of large scale graphs as well. As the name indicates, MapReduce works in two phases - a

map phase followed by a reduction phase. The system is comprised of a *master* node and a number of *workers*. In the first phase, the master node divides the input into smaller sub-problems and distributes it to all the workers. Each worker independently works on its subproblem and returns its result to the master node. In the reduce phase, all the workers combine the result in *someway* to form the answer to the original problem that the master intended.

As it turns out, the mapreduce model is not ideally suited for many different graph algorithms. e.g., if we were to do parallel BFS in MapReduce, we would need to execute the problem in multiple chained map-reduce invocations, which can be costly as it involves a lot of overhead and can subsequently lead to sub-optimal performance, poor usability and ease-of-use.

3.3 BSP - Bulk Synchronous Parallel

Bulk Synchronous Parallel (BSP) was a model proposed by Valiant [16] as a computation model for parallel processing. It was not specifically formulated for graph processing but as a promising candidate for general purpose parallel computation. It was not proposed as a programming concept or hardware but as a *bridging model* - an abstract version of a computer which provides a conceptual bridge between the physical implementation of the machine and the abstraction available to a programmer of that machine. It was defined as a combination of three core attributes [adapted for graphs from [16]] and run in a series of supersteps:

1. A number of processes running concurrently on a part(s) of the graph - performing computation and/or memory functions,

2. A communication layer that delivers messages between pairs of processes - not caring for whether they reside on the same machine or are distributed over a network, and

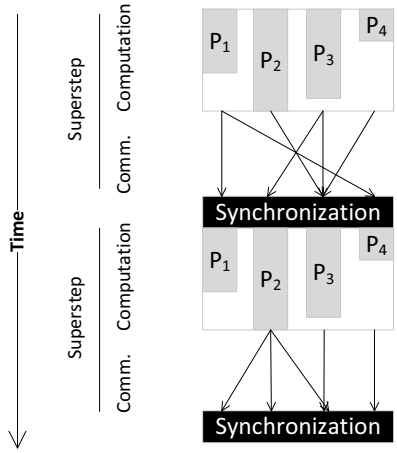


Figure 3.1: An example for Bulk Synchronous Parallel (BSP)

3. A synchronization barrier in which each processor waits for other processes to finish the processing and also to receive all the messages destined to it. Once both the conditions are satisfied, the model continues with the next superstep. If all the processes have run the supersteps and there are no messages waiting, the whole model terminates and signals the successful execution of the program as a whole.

BSP is a simple, yet efficient and scalable paradigm for parallel algorithm design and analysis. It has two performance bottlenecks: (1) synchronization has its implications in terms of performance when one process has finished, but it has to wait for all others to finish before proceeding to the next superstep. E.g., in figure 3.1, P_2 finishes much later in the second superstep, but all the other workers have to wait for it to finish. In addition, (2) BSP does not take into account the case of heterogeneous clusters, since in that case the synchronization problem is exacerbated even more and *even* distribution of jobs may overwhelm a slow process. Work has been done to study the latter limitation in [17], however they are not a focus for this paper.

3.4 Vertex-Centric Graph Processing

Vertex-Centric Graph Processing is another model proposed by Google in [18]. It is the first of its kind distributed system tailored specifically for the processing of large scale graphs. The system is basically inspired by the BSP model and since it is vertex-centric, the vertices of a graph take center stage in this system. Each vertex can best be thought of as a process (component) in BSP i.e., they are the computation unit and can use the communication layer to send and retrieve messages from other vertices. After a superstep, the messages are synchronized (i.e., received by all the vertices), then the computation can carry on with the next superstep and the process keeps repeating until the algorithm finishes.

The first manifestation of this idea came in the form of Pregel [18]. The basic architecture of Pregel is very intuitive. The system is comprised of a master and a number of workers. The input graph is partitioned and assigned *uniformly* to all the workers - no partition is assigned to master, its only role is to coordinate and oversee the worker activity. Once the graph is distributed, the master signals all the workers to start execution. At this point, worker loops through its active vertices, calling the *compute()* method for each active vertex with the messages received from the last superstep. The vertex can vote to halt if the algorithm decides the role of vertex is done and should not be called in the subsequent supersteps. An inactive vertex can be made active again with an incoming message, once all the vertices are inactive, master instructs all the workers to halt computation marking the end of a given job.

In addition to simplicity and efficiency, Pregel also offers additional abstractions like combiners and aggregators that can be exploited to further boost the performance of the system. Since, the system works on huge graphs in a distributed fashion, there is an increased likelihood that even *one* worker failure can result in the crash of whole system. To avoid this, Pregel supports fault tolerance and will dynamically reassign the job if one of the workers

fails.

Since Pregel is not available to the general public, some of the most visible alternatives that follow the same principles are: GPS (Graph Processing System) [19], Apache Giraph [20] and Apache HAMA [21].

Chapter 4

Distributed Algorithms for Graph, Dual and Strict Simulation

In this chapter, we give an outline of a distributed algorithm each for Graph, Dual and Strict simulation that are designed for a vertex-centric system. We deliberately omit Strong simulation because of limited space, plus it is similar to Strict simulation. The details of these algorithms can be found in [4].

4.1 Graph Simulation

In the designed distributed algorithm for graph simulation, the query graph is distributed among all vertices of the data graph, and then each vertex should find out its match set among the vertices of the query graph. Vertex u of the data graph matches to vertex v of the query graph if they have the same labels, and any child of v has at least one match among the children of u.

In a vertex-centric system, a vertex initially knows only about its own *label* and the *id* of its children. Therefore, each vertex needs to communicate with its neighbors to learn

Superstep 1:

- Set match flag true if there is any vertex in query with same label
  - Make a match set of potential match vertices
  - Ask children about their status
- Otherwise vote to halt

Superstep 2:

- If the flag is true reply back with the local match set
- Otherwise vote to halt

Superstep 3:

- If the flag is true evaluate the members of the local match
  - In the case of any removal from the match set, inform parents and set match flag accordingly
  - Otherwise vote to halt
- Otherwise vote to halt

Superstep 4 and beyond:

- If there is any incoming removal message reevaluate the local match set
  - In the case of any removal from the match set, inform parents and set match flag accordingly
  - Otherwise vote to halt
- Otherwise vote to halt

Figure 4.1: Summary of Graph Simulation algorithm

about their labels and status in order to evaluate the children-relationship condition. A Boolean flag, called match flag, is dedicated to each vertex which indicates if the vertex has a potential match among the vertices of the query graph. This flag is initially false. The summary of the steps of the algorithm is displayed in figure 4.1.

In an example displayed in figure 4.2, all the vertices of the data graph labeled a, b, and c make their flag true at the first superstep, and then vertices 1, 2, and 5 send messages to their children. At the second superstep only vertices 5, 6, and 7 will reply back to their parents. At the third superstep, vertices 1, 5, 6, 7, and 8 can successfully validate their match set,

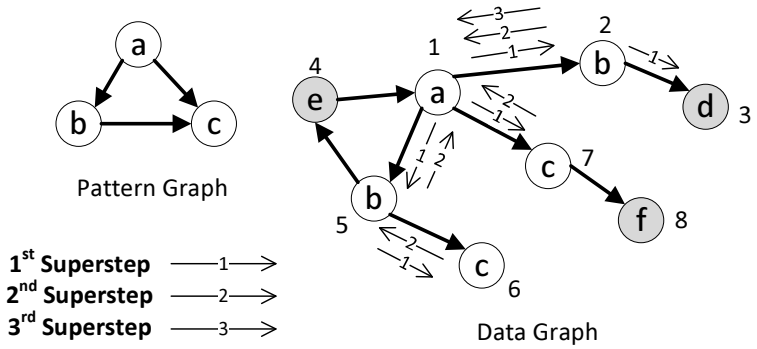


Figure 4.2: An example for distributed graph simulation

but vertex 2 makes its flag false, because it receives no message from any child. Therefore, vertex 2 sends a removal message to vertex 1. This message will be received by vertex 1 at superstep four. It will successfully reevaluate its match set, and the algorithm will finish at superstep five when every vertex has voted to halt (there is no further communication).

4.2 Dual Simulation

The algorithm for dual simulation is very similar to graph simulation. In addition to *childrenMatchSet*, we extend the algorithm by *parentMatchSet* which serves to store the match sets of all the parent vertices. Like graph simulation, we evaluate the duality condition by using the *parentMatchSet* in combination with *childrenMatchSet*. At the end of the algorithm, all the vertices with the *match* variable set to *true* are the dual match results to the query pattern.

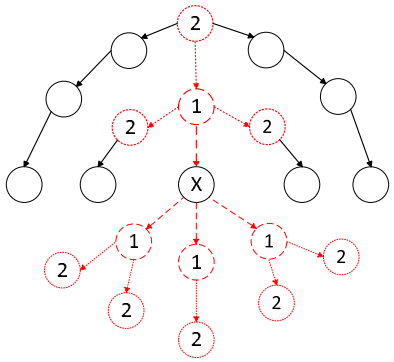


Figure 4.3: A ball around a vertex with $d_q = 2$

4.3 Strict Simulation

Strict simulation is done in two phases: (1) we run dual simulation to get the match set R and then (2) $\forall v \in R$, we create a ball with a diameter of d_q. Once we have all the balls ready, we run dual simulation on each to have the final output of strict simulation. The algorithm for strong simulation is quite similar in which we just skip the step (1) of strict simulation and start with (2) where we create the balls for every vertex in the data graph.

The biggest challenge we faced in strict simulation was the creation of balls. Because of the scale of graphs, we may end up creating balls for a lot of vertices simultaneously which could bog down the whole system. Therefore, we tried two different approaches that we go through below using Figure 4.3 as an example.

4.3.1 Depth-First Ball

As the name suggests, ball creation process works in a depth-first fashion. In the first superstep, every vertex in R sends a forward message of format *M(origin, sender, ballSize, direction)* to all of its adjacent vertices, where *origin* is the id of the center of ball (for example x), *sender* is the vertex sending out this message, *ballSize* is a value that is decremented by

one as this message propagates through the graph, and *direction* indicates if the message is going from a parent to a child or vice versa. In the second superstep, every vertex that receives a message m (for example y) will perform the following actions: (a) depending upon m.direction, it sends outs a reply message like r(*m.sender, y*) or r(*y, m.sender*) to *m.origin*; (b) it forwards a new message m_y*(m.origin, y, ballSize-1, direction)* to all of its adjacent vertices. Whenever the center of a ball receives a message which has its own id as the originator, it will add the gathered information to its ball. The messages are passed around until the value of ballSize they carry becomes 0; at that point the originator vertex has all the information of its ball.

This approach completes building the ball in *ballSize+1* supersteps, by sending out very small messages. Its major problem is the exponential growth in the number of messages because every vertex forwards its incoming messages to all its neighbors; therefore, this approach has a poor performance for balls with slightly big diameters. Our experiments also approved the problems of this approach; hence, we did not use it in our final implementation.

4.3.2 Breadth-First Ball

This approach works on a simple ping-reply model. The ball center vertex starts off by sending a ping message to all of its adjacent nodes. In the second superstep, all the recepient nodes reply back with their label and the ids of their children and parents. Center vertex upon receiving this information in the third superstep, saves the returned labels and then ping the parents/children of its parents/children. This process is repeated till we have a ball of size d_q.

For example, in figure 4.4, let us suppose we want to create a ball around vertex X of radius 2. Initially, X only knows its adjacent node ids 1,7 and no label information. It starts off by sending a ping message to all of its adjacent nodes. In the second superstep, all the recipient nodes reply back with their labels and the ids of their children and parents. Thus,

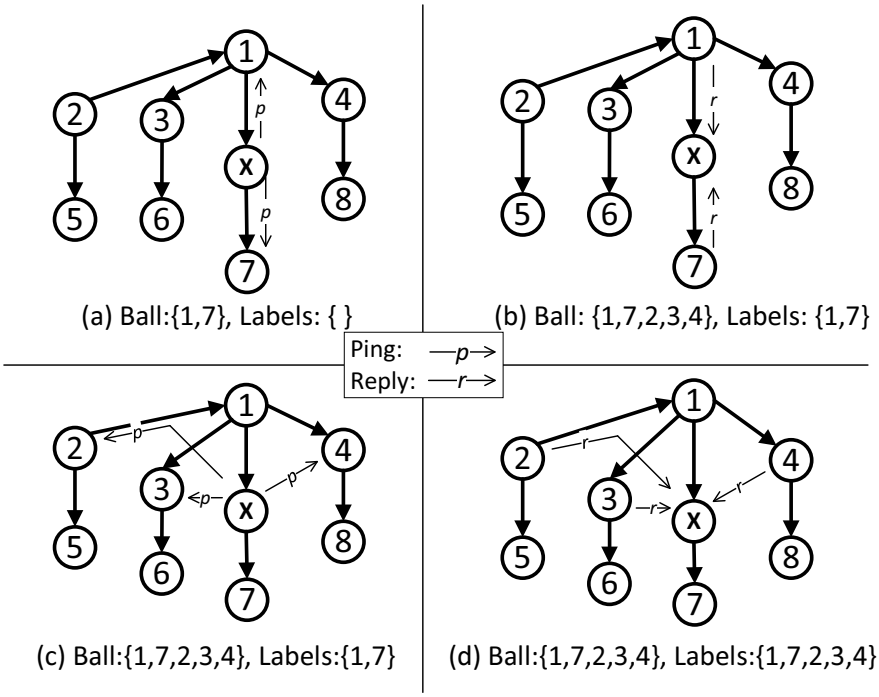


Figure 4.4: A breadth-first ball around vertex X with $d_q = 2$

in 4.4(b), X upon receiving this information saves the returned labels and the children and node ids in the ball. In 4.4(c), it sends another ping message to all of its boundary nodes which reply back with their labels and the ids of their children and parents, consequently saved by the node X in 4.4(d).

The downside of this approach is that it results in almost twice the number of supersteps, yet it was much more effective and performed way better than the other approach. Since its performance was much better than the depth-first approach, we adopted this approach for strict simulation tests.

Chapter 5

Implementation of Distributed Algorithms

Since this paper is primarily focussed on graph pattern matching using the four simulation methods discussed in Background, in this section we will discuss the implementation of graph simulation, dual simulation and strict simulation that were proposed in [4]. Since strict and strong simulation are quite similar, we have omitted strong simulation due to space constraints.

5.1 GPS - Graph Processing System

As mentioned above, because all of the four algorithms were designed with a BSP and vertex-centric model, we decided to use something akin to Pregel for the implementation. Of the numerous implementations of Pregel available, we picked GPS for our algorithms since it offers everything that we desire of Pregel and is available open-source [22]. It is written in JAVA and also gives us an option to write a *master.compute()* method that proved to be quite handy in case of strong and strict simulation.

As in Pregel, there are two types of main components in the system: one master node and k worker nodes. All the nodes communicate with each other using apache MINA which is a network application framework built on Java's asynchronous network I/O package (java.io). All the nodes run on top of HDFS (Hadoop Distributed File System) [23] which acts as the storage layer E.g,. for the input data graph

GPS offers a very simple programming model. Every vertex in the graph can be abstracted by creating a class that inherits from a *vertex* abstract class and we just need to overload its *compute* method. The compute method has two inputs: (1) an Integer *superStepNo* that represents the particular superstep number compute has been called for and (2) Iterable⟨M⟩ which is an iterator available over all the incoming messages for superstep *superStepNo*. *superStepNo* is very helpful in driving the logic of algorithm.

A GPS job starts off by partitioning the data graph over all the participating workers. Every worker reads its partition and then distribute the vertices based on the round-robin scheme i.e., vertex v gets assigned to the worker $W = v.id \ \% \ k$. The lifecycle of a GPS job can be summarized in following steps: (a) parse the input graph files (b) start a new superstep and (c) terminate computation when all the vertices have voted to halt and there are no messages in transit, otherwise go to (b).

5.2 Akka

Akka is a toolkit and runtime for building highly concurrent, distributed, and fault tolerant event driven applications on the JVM. It has an extended API that lets you manage service failures, load management (back-off strategies, timeouts and processing-isolation), both horizontal and vertal scalability (add more cores and/or add more machines) [1]. The API is available both in Java as well as Scala.

[1] http://akka.io/

5.3 GPS vs Akka

GPS provides an excellent platform for graph processing. However, to enable the message passing for custom types, requires considerable effort that is not only time-consuming but is also prone to bugs. E.g., if the message contains some complex type like Map[Int, List[Map[Int, Int]]], we need to be very careful with how the message is serialized and then deserialized at the receiving end. These nitty-gritty details can be cumbersome, hard to maintain and not so readable.

Akka is a lot different to GPS (Pregel) since it is a general purpose toolkit for building highly concurrent and distributed applications and not something that is built ground-up only for graphs. It means there is some work that needs to be done to make Akka work in a fashion similar to BSP. Perhaps, the biggest edge that Akka has over other comparable models is its inherent ability and support for sending messages between actors. With Akka (Scala), the developer does not have to worry about serializing/deserializing of data. They can send messages wrapping complex types with extremely concise and terse syntax. E.g., in figure below, we can clearly see how easy it is to exchange messages between actors using the Scala syntax for message passing. To send the message, we just wrap it inside a case class and then the receiving actor can process the message however he wishes.

```
1   // case class for message
2   case class message (...)
3
4   // code for ActorA (sender)
5   val msg: Map[Int, List[Int]] = ...
6   actorB ! message(msg)
7
8   // code for ActorB (receiver)
9   def receive = {
10    case g: message => ...
11  }
```

In parallel computing, concurrency has always been an important issue that needs to be taken care of. For quite a while, threads have been the preferred method of achieving concurrency, however it has its issues related to synchronization giving rise to problems like deadlocks, starvation, livelocks, etc. With the actor-based model (as in Akka), since the *state* is *shared* with message passing, most of these issues at an implementation level are taken care of. However, it should be noted that Akka (or any other actor model) is still susceptible to synchronization issues arising out of the algorithm itself. Also, with actors we do not have to rely on polling for information, which saves us more time and computational power.

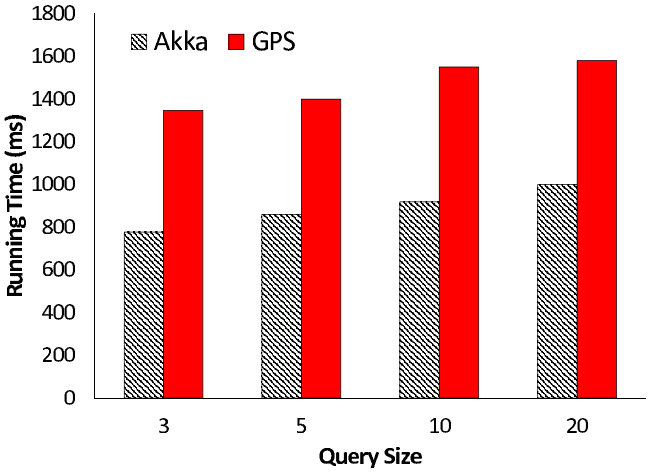


Figure 5.1: Comparison between GPS and Akka using Graph Simulation

We implemented a prototype application for graph simulation using Akka. We were pleasantly surprised by the power and ease provided by Akka and Scala for rapid development. Even with the most basic implementation, we were able to achieve much better times as we were getting from a much more accomplished system like GPS. The tests were conducted on the amazon-2008 [24] dataset and were run over a cluster of 5 machines. As can be seen in Figure 5.1, Akka ran almost twice as fast as GPS with a naive implementation and some initial testing.

Chapter 6

Performance Evaluation

In this chapter, we go through the results of experiments conducted for these algorithms on GPS.

6.1 Experimental Setup

We conducted extensive testing for all of the three algorithms i.e., graph simulation, dual simulation and strict simulation. The experiments were conducted on a cluster with 12 machines in all. Each machine has two 2GHz Intel Xeon E5-2620 CPUs, each with six cores. The ethernet connection is 1Gb/sec. Each machine is running Java 6 and Hadoop HDFS is setup on the whole cluster. We also setup GPS on all the machines since we wanted to have the ability to use all the computers in our experiments.

6.1.1 Datasets and storage

We used three datasets for our experiments - two were real-life and one was synthesized. The synthesized dataset has 10^8 vertices with around 4 billion edges. Considering the huge size of file, we broke the graph into multiple files adopting the simple adjacency list format

for storage. Every line in the file has the standard format like below:

```
{vertex-id} {vertex-label} {list of outgoing edges ids separated by comma}
      0           20         5, 192, 23, 5, 18239
      1           23
..............
..............
```

One real life dataset was uk-2005 downloaded using WebGraph and LLP [25, 26]. It is a 2005 crawl of the .uk domain performed by UbiCrawler[27]. The graph has 39459925 nodes and 936364282 edges with an average degree of 24.

The other real life dataset was enwiki-2013. The graph represents a snapshot of the English part of Wikipedia as of late February 2013 [24] vertices equalling 4206785 nodes and 101355853 edges. It is a more dense graph than uk-2005.

Unless mentioned otherwise, the labels on all the vertices were randomly chosen as an integer in the range from 0 to 200. All of these graphs were loaded into the Hadoop HDFS for GPS consumption.

We generated queries using two parameters, (1) the number of vertices $|V_q|$ in the query, and (2) α_q which controls the edge density of the graph and is used as $|V_q|^{\alpha_q}$.

6.1.2 Results Verification

To verify the implementation and the results, we put together a combination of few tools with the workflow shown in Fig 6.1. The process begins by automatically generating a synthesized data graph of random size and then generating a random query out of it. Once it has both the inputs, i.e., the data graph and query graph, it runs the sequential as well as the distributed version of the algorithm and then the results are compared. If equal the process is repeated, otherwise the whole thing halts and the system reports the fault.

We ran this comparison for more than a 1000 times for all the three cases of graph, dual and strict simulation. Once we verified the implementation using this method, we proceeded to the experiments section.

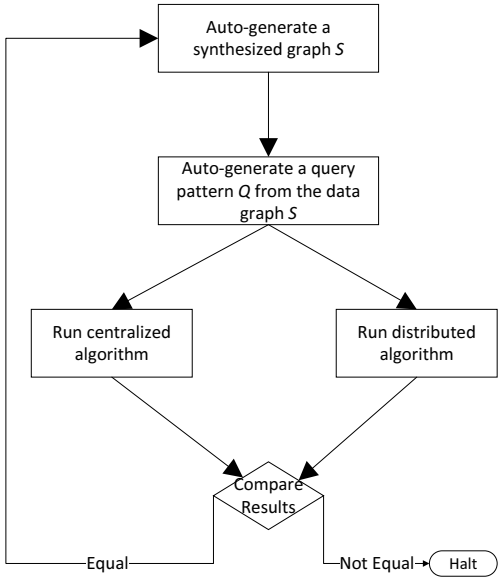
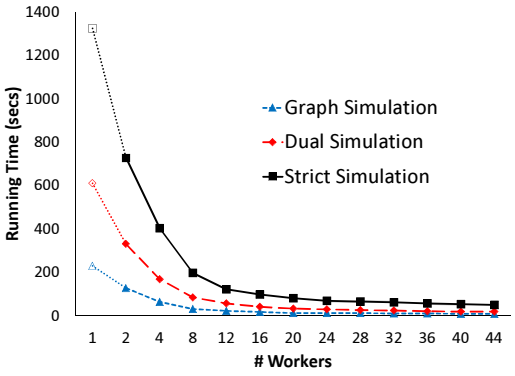


Figure 6.1: The workflow of the auto comparison model

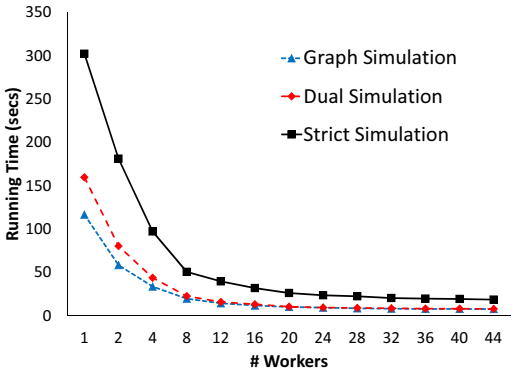
6.2 Experimental Results

We wanted to do a detailed study of how the algorithms behave with different input variations. In this report, we only focus on the speedup and efficiency of the algorithms, a more detailed report can be found in [4].

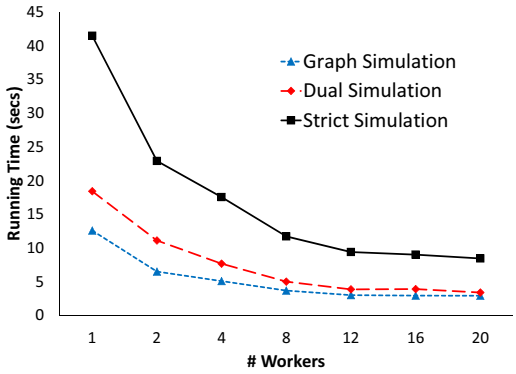
Before delving into the details of speedup and efficiency, we show the running times of the three algorithms on the three datasets.



(a) Synthesized Dataset, $|V| = 10^8, \alpha = 1.2$



(b) uk-2005, $|V| = 3.9\text{x}10^7$



(c) enwiki-2013, $|V| = 4.2\text{x}10^6$

Figure 6.2: Running times of graph, dual and strict simulation, $|V_q| = 25, \alpha_q = 1.2$

6.2.1 Speedup

The speedup is how much faster the parallel program is, for instance if T_p is the time it takes to solve the task on p processors, then speedup $= T_1/T_p$. For our algorithms, we adapt the term speedup as how the distributed algorithm gets faster as we increase the number of workers. We can use workers to calculate speedup since they represent the *logical* processors

for our system. Even, when we ran multiple workers on the same physical node we made sure that we are not exhausting the number of available threads, so each worker could run as indepedently as possible. Plus, all the physical nodes are the same.

We show the results of our experiments on the different datasets for the corresponding speedups in Fig 6.3, 6.4 and 6.5. In all of these experiments, we used 11 nodes as workers and the remaining one node as the master. On the x-axis, we have the total number of workers and the other axis corresponds to the speedup. It should be noted that we have more workers than the number of worker node machines available. We achieve this by distributing the workers over the 11 worker nodes available. For instance, if we have 44 workers in all we are essentially running 44/11=4 workers per node.

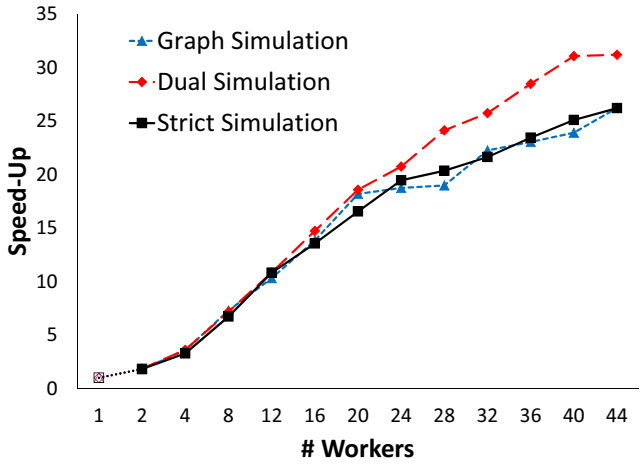


Figure 6.3: The speedup on the synthesized dataset, $|V| = 10^8$, $|V_q| = 25, \alpha = \alpha_q = 1.2$

We can easily observe the following from the three figures:

1. As we increase the number of workers, we get more speedup. However, the speedup bar tends to flatten as we approach the maximum number of workers. We can see that a speedup of more than 11 - which were the total number of worker machines

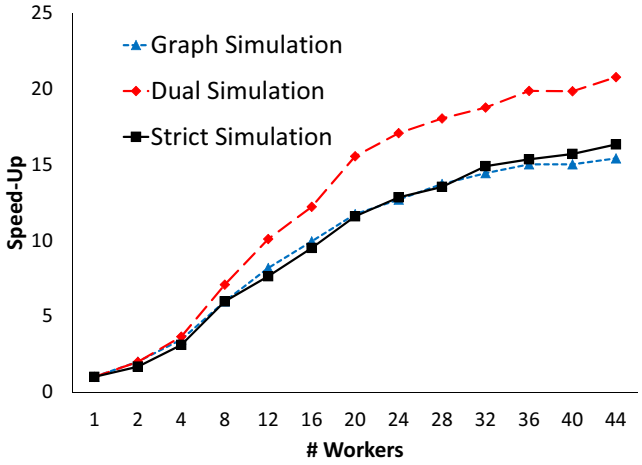


Figure 6.4: The speedup on the uk-2005 dataset, $|V| = 3.9\text{x}10^7$, $|V_q| = 25, \alpha_q = 1.2$

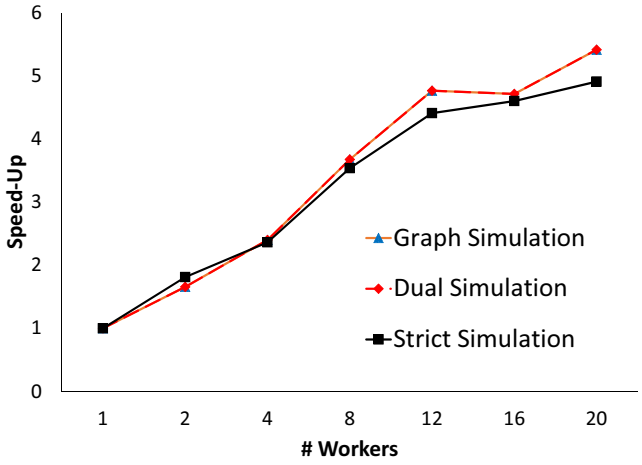


Figure 6.5: The speedup on the enwiki-2013 dataset, $|V| = 4.2\text{x}10^6$, $|V_q| = 25, \alpha_q = 1.2$

available - was achieved in the synthesized as well as uk-2005 dataset. The relatively flat behavior at the higher end is because the input job has been divided into very

tiny sub-problems, thus each worker has a smaller chunk of the original problem and the synchronization costs of all the workers become an important factor in the total computation time.

2. We get more speedup with bigger datasets, and lesser speedup with smaller datasets. The behavior can be partially explained with the point above. If the input dataset is already small (e.g., Fig 6.5), then even by introducing more workers, we will not be able to see the same gain as we would see from a bigger dataset. In fact, the curve can drop down because of the synchronization and communication costs.

3. It must be noted that we could not fit the synthesized data on a single worker. Based on multiple experiments spanning small as well as bigger datasets, on average we achieve a speedup of 1.8 as we jump from 1 to 2 workers, therefore we have extrapolated that much speedup (indicated by the dotted segment), had we been able to get it running on a worker with enough memory. We can see that we get more than 30x speedup on dual simulation which is quite impressive considering we are only running 11 worker nodes.

6.2.2 Efficiency

Efficiency is how effectively additional processors are used in a distributed system, Efficiency = Speedup/p, where p are the number of processors. Just like speedup, we can use the workers as our processors since they effectively represent the *logical* processors in our system. In essence, we try to get an idea of the system potential. We present the results of our efficiency experiments on the three datasets in Figures 6.6, 6.7 and 6.8.

We can observe the following from the efficiency graphs:

1. Just like any other comparable model, the efficiency drops as we increase the total number of workers. However, the drop is less sharp in the case of bigger datasets.

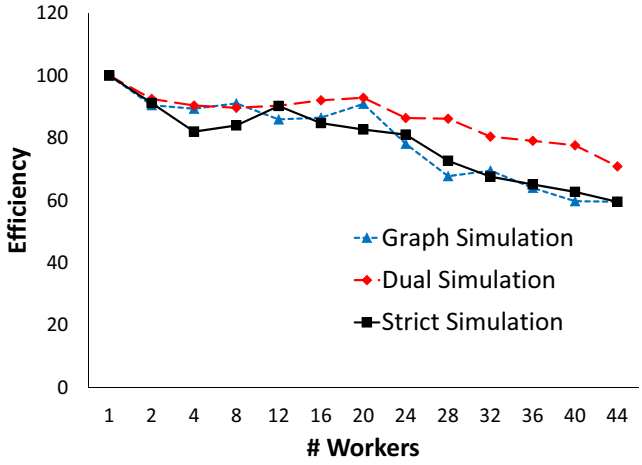


Figure 6.6: Efficiency for the synthesized dataset, $|V| = 10^8, |V_q| = 25, \alpha = \alpha_q = 1.2$

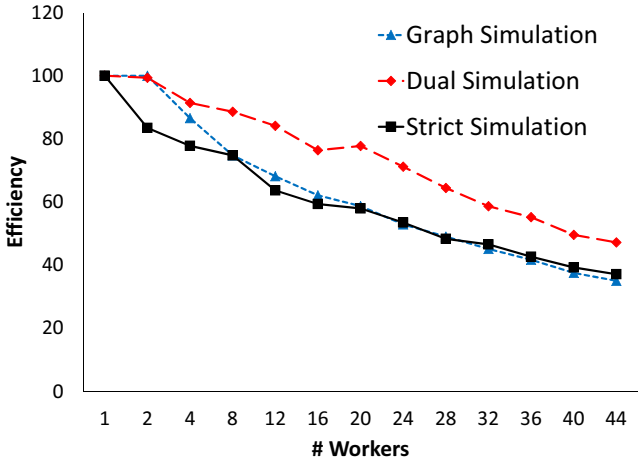


Figure 6.7: Efficiency for the uk-2005 dataset, $|V| = 3.7\text{x}10^7$, $|V_q| = 25, \alpha_q = 1.2$

2. With smaller datasets, we drop below 50% efficiency with just 8 workers (e.g., fig 6.8). This is because when the dataset is small and we distribute it to many workers, the

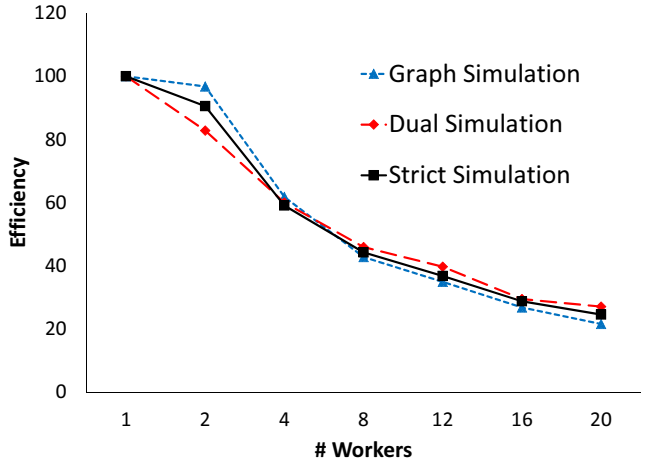


Figure 6.8: Efficiency for the enwiki-2013 dataset, $|V| = 4.2\text{x}10^6$, $|V_q| = 25, \alpha_q = 1.2$

synchronization and communication costs in the system become the significant factor in the total running time.

Chapter 7

Impact of Graph Partitioning

In this section, we try to study the effects of graph partitioning as an optimization to the existing algorithms. Basically, we want to study how the algorithms react to different partitioning schemes - and if it results in any improvements. Graph partitioning has extensive applications in many areas including telephone network design, VLSI design and task scheduling. The problem is to partition the graph into p roughly equal parts, such that number of edges connecting vertices in different parts is minimized [28]

It is important in the context of distributed computing because we want to partition the graph into pieces such that each piece is mostly *self-contained* i.e., we want to reduce its communication to other parts as possible, thus in theory resulting in speed-up. However, it is not a trivial problem. When we partition the graph, we need to take care of two points:

1. We want to minimize the number of edges going from one partition to the other,

2. An effort must be made to partition the graph into equal parts, so every worker gets a fair amount of load. This is essential because we do not want to distribute the jobs unevenly, thus losing out on the benefits of parallelism.

Graph partitioning that enforces the two conditions above is called *min-cut* partitioning

which is a NP-complete problem and has been extensively studied in its own right. Graph partitioning has been successfully used with considerable improvement in various applications. It must be noticed that graph partitioning has no guarantees to provide consistent improvements for all graph algorithms. It will result in most speed-up, if the communication is mostly between adjacent vertices, since partitioning tries to co-locate them in a single partition, however if that is not the case then communication is inevitable. Semih [19] was able to achieve 2.2x improvement in speed by partitioning with PageRank algorithm, but in case of Highly-Connected Component, Single Source Shortest Path the speed-up was only 1.47 and 1.08 respectively.

Since GPS default partitioning method works on a simple hash function (as described in section 5.1), we call it *round-robin* (rr) partitioning to differ it from some other methods we test. Now, we present our intuition as to how graph partitioning should work in our case and then back it up with experiments and results in the next section.

Since graph partitioning takes into account the topology of graph we are prone to get imbalanced load among workers as the algorithm runs which can harm the overall performance.

Example

Let us walk through the example shown in figure 7.1 with two hypothetical workers W_1, W_2 and suppose we are running strict simulation. The query and data graphs are given in the figure 7.1, with the resultant subgraph highlighted in the data graph. The vertex labels are inside and the *id* is hanging outside the node. In the default case, the system will employ a simple round-robin scheme to distribute the vertices over workers. We have highlighted the vertices that go to worker W_1 in grey and the ones in white go to W_2. As we know strict simulation runs in two phases - dual simulation followed by the ball creation process. We

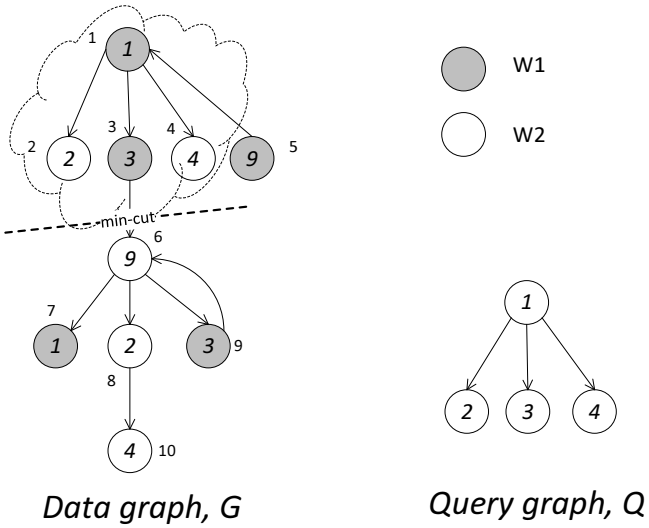


Figure 7.1: Graph partitioning

can clearly see that the matched subgraph is evenly distributed over both the workers and the balls are created by both the workers, thus maximum parallelism is achieved.

Now, let us consider the min-cut partitioned case. We have marked the edge with *min-cut* where the graph will be partitioned. Now, one of these partitions will go to worker W_1 and other will go to W_2. In this case, we can clearly see that the whole matched subgraph will be contained in one worker and only that worker will be creating the balls, thus essentially the algorithm is run in a serial fashion, resulting in slower times.

Based on the example above, we can say that with queries having a lower α_q (denotes the number of edges, used as $|V_q|^{\alpha_q}$), there is an increasing probability that we get better speedup as there is an increasing chance that the resultant matches will be distributed over multiple workers, thus we will be able to enjoy the benefits of parallelism. However, on the other hand with queries having a higher α_q, we have more chances that the resultant subgraph(s) will not be well distributed. These issues can become quite visible in algorithms like strict

simulation where some operations (like creation of balls) are expensive and if they are local to fewer workers, it can tremendously harm the overall performance of the algorithm, as will be shown by our results in the next section.

Performance Evaluation

We conducted extensive testing of graph partitioning on multiple datasets and in this section, we try to present the results and reason about them.

Datasets

We use both a real world dataset and a synthesized dataset. Data graph is governed by three variables (1) $|V|$ is the number of vertices, (2) α is the number of edges used as $|V^{\alpha}|$, and (3), l is the number of labels in the graph that are randomly assigned from the range (0, l). For query graphs, the only parameter used is $|V_q|$ indicating the number of vertices and α_q, if not mentioned otherwise is kept constant at 1.2.

uk-2002 [24] was used as the real life dataset, that is a 2002 crawl of the .uk domain. It has 18520486 vertices and 298113762 edges. For the synthesized data graph, we used the same synthesizer we used in [4] to generate a graph with $|V| = 10^7$, an α of 1.2 and l of 200.

The synthesized dataset had 10^7 vertices with an average degree confirming to $\alpha = 1.2$. It was generated using our own synthesizer and the graph had random edges between the vertices.

To retrieve a query graph for a particular data set, we took $|V_q|$ as the input which is the size of query graph. We also took α_q as input which governs the edge density. α_q, if not mentioned otherwise is kept constant at 1.2. Once, we have both the arguments then we randomly extract a connected subgraph from the dataset that adheres to the two conditions

and will be subsequently used as the pattern graph.

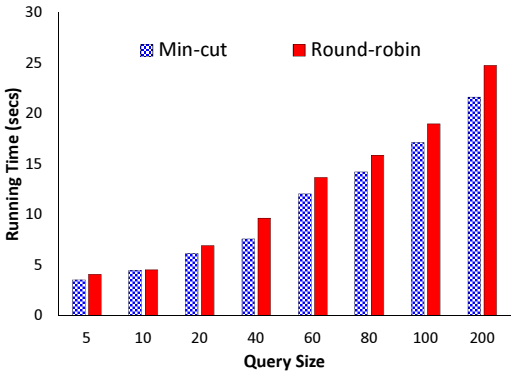
Experimental Environment

The experiments were conducted using GPS on a cluster of 5 machines. Each one has a 128GB DDR3 RAM, two 2GHz Intel Xeon E5-2620 CPUs, each with 6 cores. The ethernet connection is 1Gb/sec. Four machines acted as slaves with one as the master. Every experiment was repeated a total of three times and then the average is reported in the results section below.

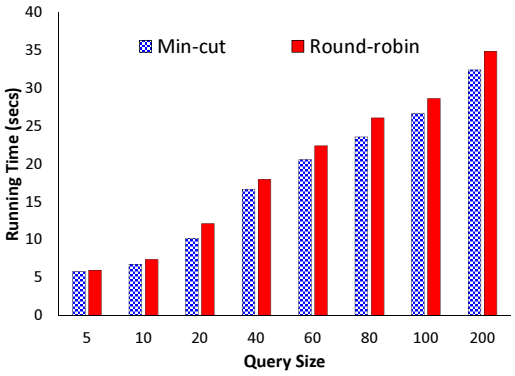
7.1 Experimental Results

We used METIS [28] for graph partitioning, which can partition an unstructured graph into k parts using either the multilevel recursive bisectioning [29] or the multilevel k-way [30] schemes. Both the models can provide high-quality yet different partitions so we tried both to study the relative impact. The algorithms work with a simple goal: *edge-cut* which basically tries to minimize the edges that travel between different partitions. Since graph partitioning is not the focus of this report, we do not discuss the details of the two types here. After extensive testing on both (k-way as well as recursive bisectioning), we found that k-way performed better on average, therefore we only report its results below. However, we do present the results of both the partitionings in the appendix.

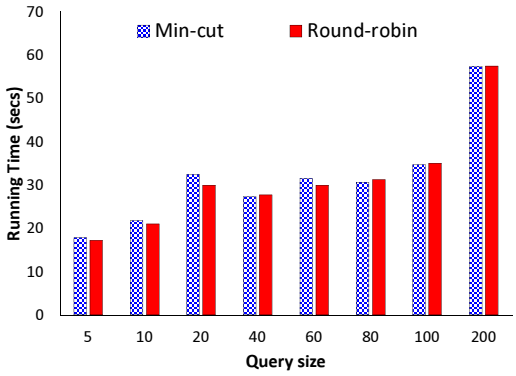
We identify two complexity measures for our tests, (1) *runtime* which is the time taken to complete a given job and (2) *network traffic* which are the number of bytes sent among workers to complete a given job.



(a) Graph Simulation



(b) Dual Simulation

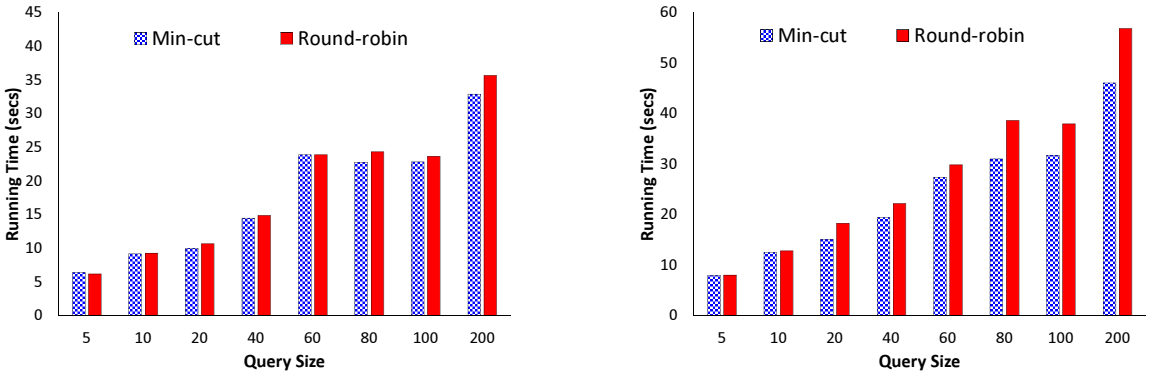


(c) Strict Simulation

Figure 7.2: Partitioning effect on the runtime of synthesized dataset, $|V| = 10^7, \alpha = 1.2$

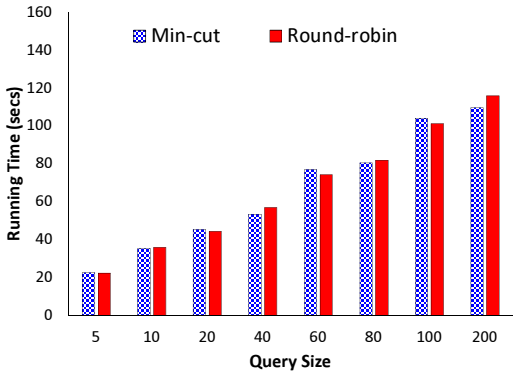
7.1.1 Runtime

With the growing scale of graphs, researchers are always trying to minimize the runtime of the algorithms. The runtime for the two datasets are presented in figures 7.2 and 7.3. On the x-axis, we manipulate the size of query and y-axis show the runtime in seconds. We run each test for a min-cut partitioned as well as the default (round-robin) case to see the



(a) Graph Simulation

(b) Dual Simulation



(c) Strict Simulation

Figure 7.3: Partitioning effect on the runtime of uk-2002-hc, $|V| = 1.8\text{x}10^7$

comparison. From the figure, we observe that:

1. The runtimes for graph and dual simulation are always faster with min-cut partitioning. In the worst case, they are about the same as the round-robin case.

2. The total runtime of the algorithm increases as we keep increasing the size of query.

3. We get the most speed-up in case of dual simulation.

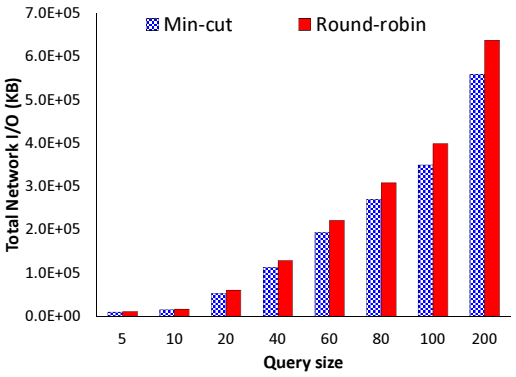
4. The runtimes are much closer in strict simulation and appear mostly unaffected by min-cut partitioning.

Graph and dual simulation runtimes take benefit from graph partitioning since the vertices always talk to their adjacent vertices only. However, strict simulation is different because when it is in the process of building the ball, it needs to communicate with vertices that are further and further away from the center. This increases the probability that the vertex will communicate with a vertex that lies on some other partition, thus washing away any benefit that we obtained from partitioning. Also since ball creation is an expensive process, even a slight imbalance in the number of vertices creating the balls, can slow down the whole system. With all of these factors, we can easily see that the times between different types of partitionings for strict simulation remain the same. It must be noted that we fixed the α_q value to 1.2 to generate queries with low-degree for all of these experiments and the results are inline with our expectations mentioned earlier.

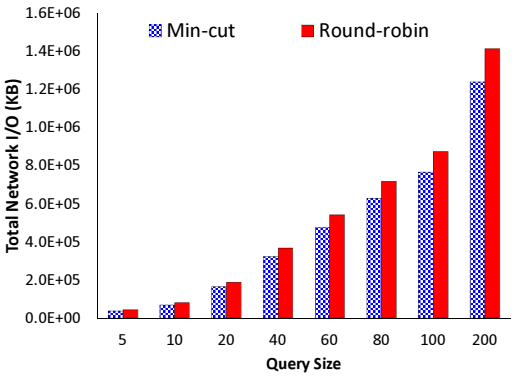
Further, since it is unfair to report the times based on a single query for any given size, we created 10 queries for each query size and repeated the experiments. Our times with those many queries were consistent with the results reported above.

7.1.2 Network traffic

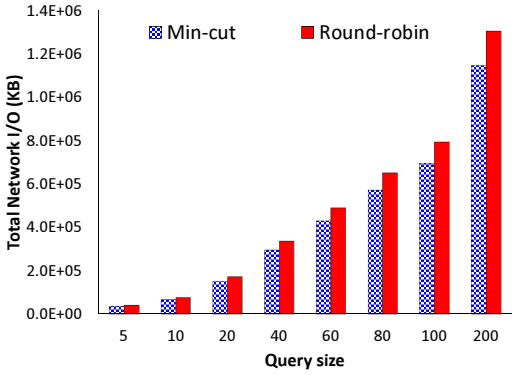
Another important criteria is the data that needs to be moved among workers. By reducing the total network traffic, we increase our chances of reducing the runtime and any cost associated with it. The effect of reduced data traffic could be more prominent in case of a constrained or a geographically distributed network.



(a) Graph Simulation



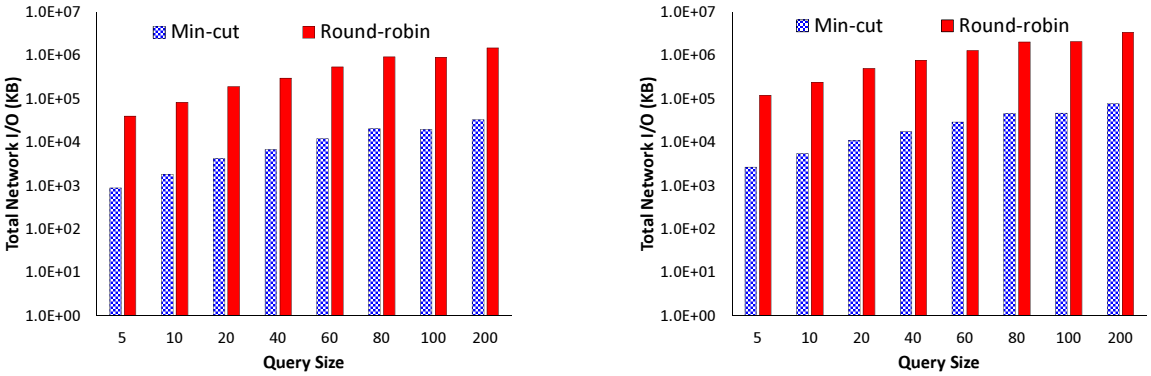
(b) Dual Simulation



(c) Strict Simulation

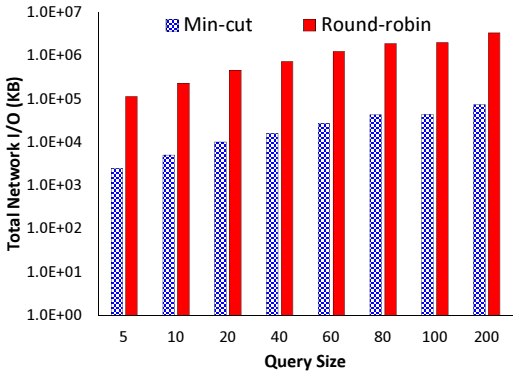
Figure 7.4: Partitioning effect on the network I/O of synthesized dataset, $|V| = 10^7, \alpha = 1.2$

In figures 7.4 and 7.5, we show the results of partitioning on the total network traffic.



(a) Graph Simulation

(b) Dual Simulation



(c) Strict Simulation

Figure 7.5: Partitioning effect on the network I/O of uk-2002-hc, $|V| = 1.8\text{x}10^7$

We can easily observe the following:

1. The network traffic for graph, dual and strict simulation is always lower with partitioning.

2. On the real life dataset (uk-2002), the network traffic drops considerably. It should be

noted that the graphs are drawn on a logarithmic scale for the uk-2002 dataset.

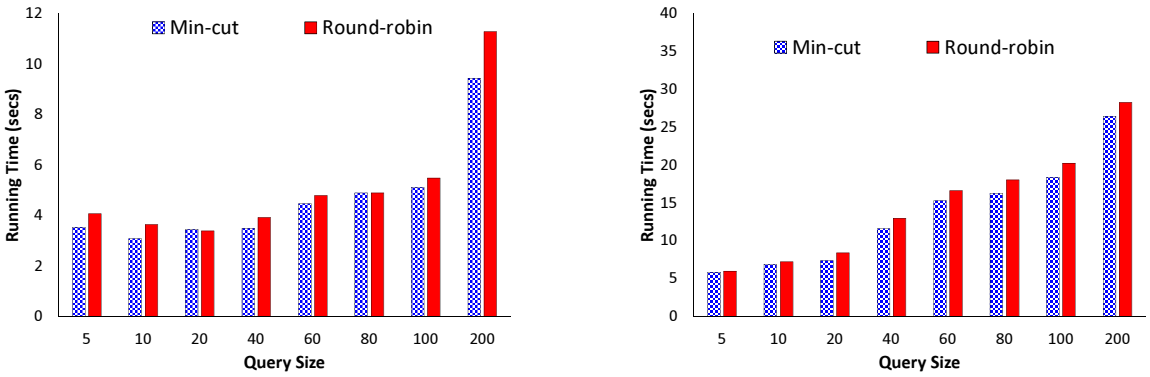
3. The total network traffic increases linearly as we increase the size of query.

The power law nature of the real-life dataset is the contributive factor to the drastic drop in the network traffic [31]. A similar pattern is not present in the synthetic dataset since it is a graph with randomized edges i.e., the vertices and edges are very uniformly distributed.

To further cement our results above, we created 10 queries for each query size and repeated the experiments. Our results with those many queries were consistent with the results reported above.

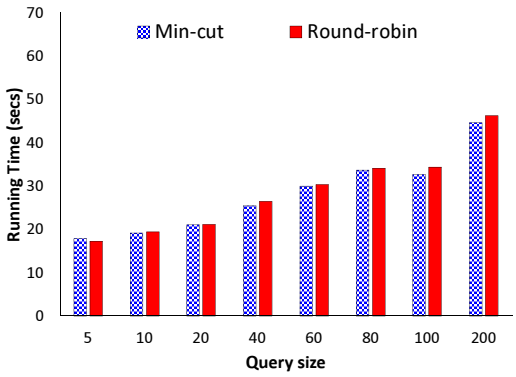
7.1.3 Diameter of pattern graph

In this section, we try to evaluate the claims made earlier with regards to the impact of α_q in conjunction with graph partitioning. Again, we conduct the tests on the two datasets used above, but this time we generate queries with an α_q value of 1.75. We have copied the results below in Figures 7.6 and 7.7.



(a) Graph Simulation

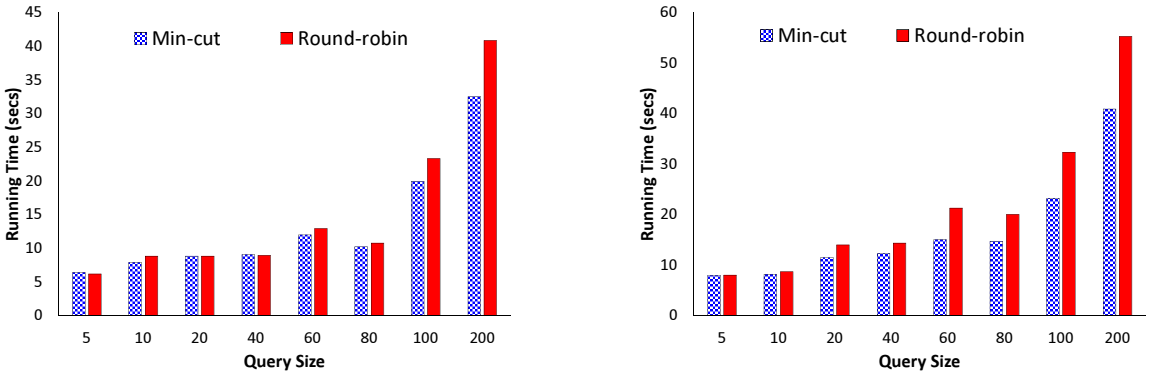
(b) Dual Simulation



(c) Strict Simulation

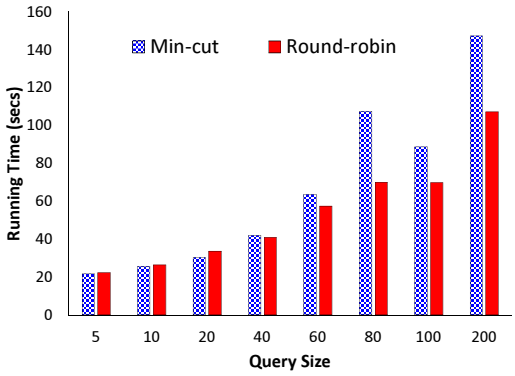
Figure 7.6: Partitioning effect on the runtime of synthesized dataset, $|V| = 10^7$, $\alpha_q = 1.75$

By going through the figures, we can make the following conclusions:



(a) Graph Simulation

(b) Dual Simulation



(c) Strict Simulation

Figure 7.7: Partitioning effect on the runtime of uk-2002-hc, $|V| = 1.8\text{x}10^7$, $\alpha_q = 1.75$

1. As the query size increases, the running time of all the algorithms increases.

2. If we compare the times against 7.2 and 7.3, we can see that these queries take less time. That is because with min-cut partitioning we are reducing the network I/O and also since we have a smaller diameter for the query graph resulting in fewer supersteps and hence we have a faster running time.

3. In case of larger queries we get faster times in both graph and dual simulations.

4. We get some interesting results in case of strict simulation with the real life dataset. To understand the reason, we need to revise the two important phases of strict simulation - in the first phase, we run dual simulation to get the result and then we create a ball on every vertex in the result. If in any case the result of dual simulation is not uniformly distributed over all workers, then some workers are forced to do much more work than others for creating balls. This slows down the whole process as ball creation is a really slow process and the biggest bottleneck in strict simulation. In the example graph where the times are much slower, we found out that all the balls were being created by fewer workers (most workers were sitting idle), thus essentially it was a sequential process in the second phase of strict simulation. Hence, these results vindicate our expectation that queries with a higher α_q, on average, will operate badly with strict simulation as compared to a lower α_q.

We repeated these tests using multiple queries (with $\alpha_q = 1.75$) and on average, min-cut partitioned datasets were slower than round-robin in the case of strict simulation. We give a snapshot of these results in Tables 7.1 and 7.2. Clearly, we can see that the times with a lower value of α_q are much more consistent and become erratic as the α_q gets larger.

Table 7.1: Runtimes of strict simulation on 10 different queries, $|V_q| = 100, \alpha_q = 1.75$

Query #	k-way	recursive	round-robin
1	54.97	56.07	54.95
2	59.62	59.85	63.49
3	88.81	92.84	69.87
4	44.15	42.72	49.80
5	81.18	75.46	57.99
6	347.72	343.37	154.67
7	198.08	215.42	105.90
8	47.59	48.79	57.05
9	96.30	94.95	74.46
10	100.20	111.09	80.75

Table 7.2: Runtimes of strict simulation on 10 different queries, $|V_q| = 100, \alpha_q = 1.2$

Query #	k-way	recursive	round-robin
1	71.78	73.60	72.75
2	73.55	73.04	73.73
3	79.39	81.91	88.64
4	73.15	78.38	75.55
5	69.64	73.06	69.53
6	68.19	68.18	65.51
7	88.42	87.22	91.16
8	76.12	80.21	83.53
9	79.75	82.04	88.46
10	59.74	64.93	61.45

Chapter 8

Related Work

The problem of subgraph pattern matching is a widely studied topic. In this section, we try to summarize the literature along three lines (a) distributed graph pattern matching with a focus on graph simulation (2) the effects of graph partitioning on distributed pattern matching algorithms and (3) a review of the different implementation options available for distributed graph algorithms.

Over the years, many different graph pattern matching techniques have been proposed. There are two broad categories of matching, *exact* and *inexact* algorithms[11]. Exact matching works by preserving the edge connectivity i.e., if the two nodes in the pattern graph are connected by an edge, they must map to two nodes in the other graph that are connected by an edge as well. There are many variations within exact matching like Graph Isomorphism, Subgraph Isomorphism, Homomorphism and Maximum Common Subgraph (MCS). Subgraph Isomorphism is possibly one of the most well-renowned technique used for pattern matching. However it is a NP-complete problem which matches the graph based on the exact topological structure of pattern.

Inexact matching techniques relax the stringent conditions of exact matching and try to get the semantic matches in a graph. This *induced* tolerance in the matching process means

that even if there is a slight noise, it will give some result whereas exact matching will turn up with nothing. A few techniques like p-homomorphism [32] and bounded simulation [33] that rely on inexact but semantic matching [5] have been proposed recently.

The ideas of dual and strong simulation were introduced in [10] yet the first distributed approach for graph simulation was introduced in [34]. However, their approach is not based on a vertex-centric model and uses a modified version of the hhk algorithm. They identify three complexity measures namely visit time, makespan and datashipment. They successfully demonstrate that the distributed model has applications for large scale graph processing. However, not all of its stages are in parallel and one is strictly serial.

Graph partitioning is another problem that is gaining traction with the advent of distributed algorithms. These algorithms mostly try to work with the goal of minimizing the edge-cut in the graph. METIS [28] is a renowned tool that can employ two different techniqeus to generate high-quality partitions. The effects of graph partitioning on different algorithms have been studied by numerous researchers. [19] talks about the impact of partitioning on algorithms like PageRank, SSSP, etc. and report a noticeable speed-up. We could not find any literature that studies the impact of partitioning in conjunction with semantic web matching.

Chapter 9

Conclusion

Graph pattern matching has been an important topic in the field of Computer Science and has been gaining prominence recently. It has become more challenging with the rapidly increasing size of graphs. In this report, we study the three polynomial type pattern matching techniques in detail. Following are the chief contributions of this report:

1. We showed that the three distributed algorithms - one each for graph, dual and strict simulation, show speedup as we increase the number of workers.

2. We showed that the efficiency curve drops as we increase the number of workers. With bigger datasets, the efficiency is much higher than smaller datasets.

3. We demonstrated through experiments that min-cut graph partitioning improves the runtime of the graph simulation and dual simulation algorithms consistently as compared to round-robin distribution. The improvement in runtime becomes better as we increase the size of query graph.

4. We demonstrated through experiments that min-cut graph partitioning performs almost the same as simple round-robin scheme on strict simulation with a lower value of α_q (denote the number of edges in a graph, used as $|V_q|^{\alpha_q}$ - greater α_q means a

dense graph and lower means vice versa). However, it performs much worse than the round-robin distribution in case of query graphs with a higher value of α_q.

5. We showed that min-cut graph partitioning always result in a drop in the network I/O among workers. This can be significant in the case when the workers are distributed geographically and/or have choked bandwidth. The drop is much more significant in case of real life datasets (x100 times).

6. An overview of the different techniques that can be used to build balls around a vertex in a distributed vertex-centric setting.

We successfully show that the Pregel like vertex-centric model gives us an impressive speedup (more than 30x times) as well as efficiency on massive graphs. We also show that graph partitioning can be used to drastically reduce the data communication size, especially in real life datasets that have the power law graph attribute. Also we show that we achieve some speed-up in graph and dual simulation with min-cut graph partitioning as compared to the default round-robin distribution of vertices. Min-cut partitioning has practically no effect on strict simulation, however under some circumstances it can perform much worse than round-robin distribution.

9.1 Future Work

1. We intend to look into different ways to improve the strict simulation running times on GPS. It is clear that creating the ball is a slow and resource consuming process; hence, we need to find out how we can improve the process of ball-creation. Instead of creating the ball on every vertex, we intend to come up with techniques to reduce the number of balls without compromising the results.

   Another possible option is to do graph exploration on the dual simulation match to

prune the result set without creating the balls altogether. This can have drastic improvements on the running time of strict simulation.

2. Try to come up with new algorithms that do not suffer the synchronization bottlenecks of the BSP model, thus achieving maximum parallelism.

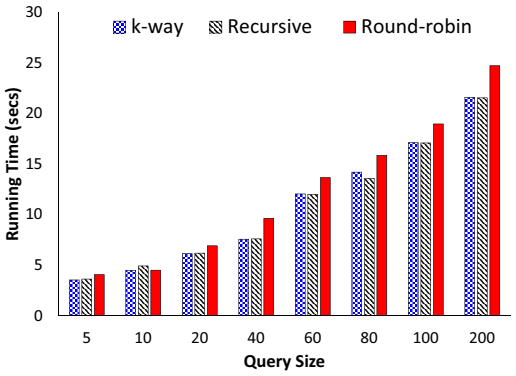
Appendix A

Results of experiments on different partitioning schemes

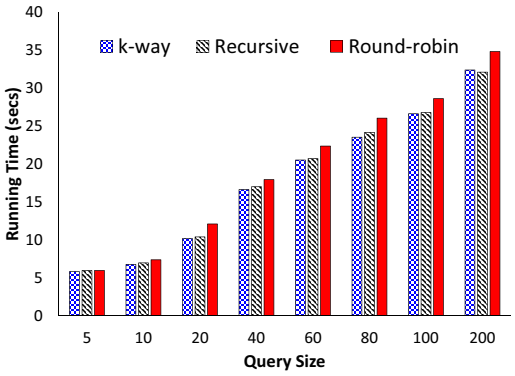
Below, we copy the results of our graph partitioning experiments using all three types of distribution of vertices - the default round-robin, and min-cut partitioning using both k-way and recursive bisectioning techniques.

A.1 Runtime

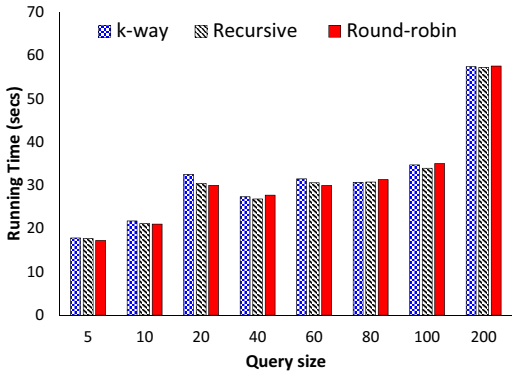
Runtime results of both datasets (synthesized and uk-2002) with different partitions are shown below in figure A.1 and A.2.



(a) Graph Simulation



(b) Dual Simulation

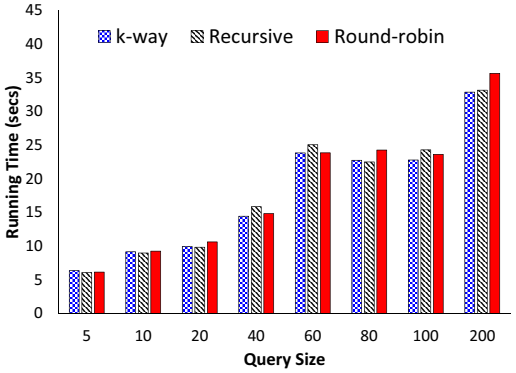


(c) Strict Simulation

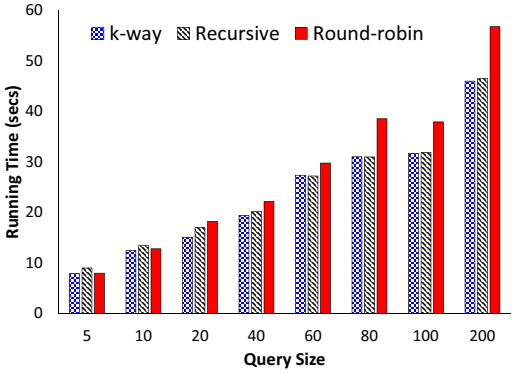
Figure A.1: Comparison of partitioning effects on the runtime of synthesized dataset

A.2 Network I/O

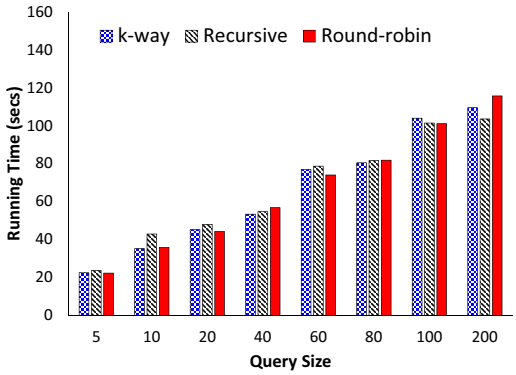
Network I/O results of both recursive bisectioning and k-way min-cut partitioning along with round-robin on both datasets is given below in fig A.3 and A.4.



(a) Graph Simulation

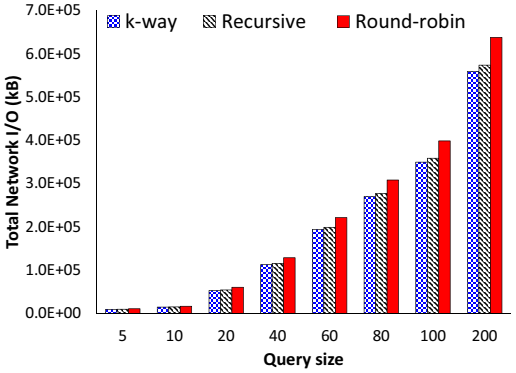


(b) Dual Simulation

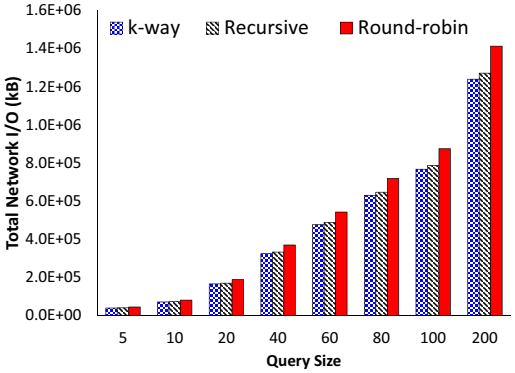


(c) Strict Simulation

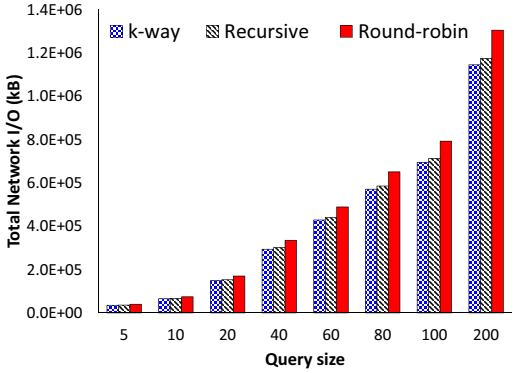
Figure A.2: Comparison of partitioning effects on the runtime of dataset uk-2002-hc



(a) Graph Simulation

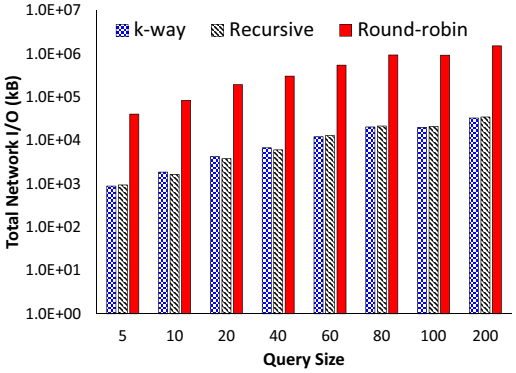


(b) Dual Simulation

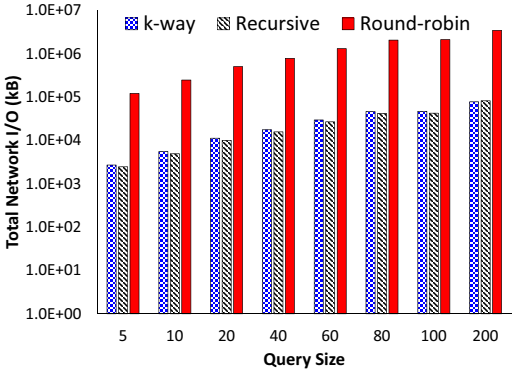


(c) Strict Simulation

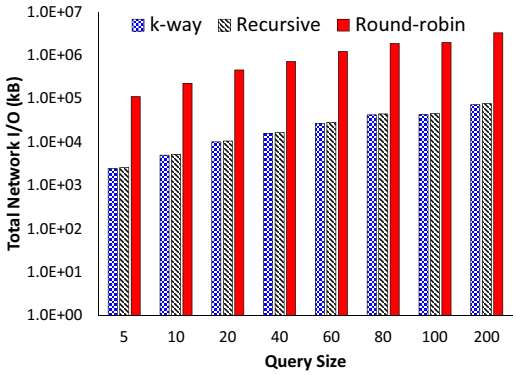
Figure A.3: A comparison of partitioning effects on the network I/O of synthesized dataset



(a) Graph Simulation



(b) Dual Simulation



(c) Strict Simulation

Figure A.4: A comparison of partitioning effects on the network I/O of uk-2002-hc

Bibliography

[1] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. ACM, 2007, pp. 29–42.

[2] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 788–799, 2012.

[3] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke, "Computing simulations on finite and infinite graphs," in *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*. IEEE, 1995, pp. 453–462.

[4] A. J. Fard, M. U. Nisar, J. A. Miller, and L. Ramasawamy, "A scalable vertex-centric approach for distributed graph pattern matching," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, 2013, p. submitted for publication.

[5] B. Gallagher, "Matching structure and semantics: A survey on graph-based pattern matching," *AAAI FS*, vol. 6, pp. 45–53, 2006.

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

[7] J. Brynielsson, J. Hogberg, L. Kaati, C. Mårtenson, and P. Svenson, "Detecting social positions using simulation," in *Advances in Social Networks Analysis and Mining (ASONAM), 2010 International Conference on.* IEEE, 2010, pp. 48–55.

[8] A. Fard, A. Abdolrashidi, L. Ramaswamy, and J. A. Miller, "Towards efficient query processing on massive time-evolving graphs," in *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2012 8th International Conference on*, oct. 2012, pp. 567 –574.

[9] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de bruijn graphs," *Genome research*, vol. 18, no. 5, pp. 821–829, 2008.

[10] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, "Capturing topology in graph pattern matching," *Proceedings of the VLDB Endowment*, vol. 5, no. 4, pp. 310–321, 2011.

[11] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Thirty years of graph matching in pattern recognition," *International journal of pattern recognition and artificial intelligence*, vol. 18, no. 03, pp. 265–298, 2004.

[12] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.

[13] D. Gregor and A. Lumsdaine, "The parallel bgl: A generic library for distributed graph computations," *Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[14] A. Chan, F. Dehne, and R. Taylor, "Cgmgraph/cgmlib: Implementing and testing cgm graph algorithms on pc clusters and shared memory machines," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 81–97, 2005.

[15] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[16] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[17] T. L. Williams and R. J. Parsons, "The heterogeneous bulk synchronous parallel model," in *Parallel and Distributed Processing*. Springer, 2000, pp. 102–108.

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

[19] S. Salihoglu and J. Widom, "Gps: A graph processing system," 2012.

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

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

[22] "GPS Source," https://subversion.assembla.com/svn/phd-projects/gps/trunk/.

[23] "Hadoop - HDFS," http://hadoop.apache.org/.

[24] "Laboratory for Web Algorithmics," http://law.di.unimi.it/datasets.php.

[25] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[26] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 2011.

[27] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Software: Practice & Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[28] G. Karypis and V. Kumar, "Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0," 1995.

[29] ——, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[30] ——, "Parallel multilevel series k-way partitioning scheme for irregular graphs," *Siam Review*, vol. 41, no. 2, pp. 278–300, 1999.

[31] B. A. Huberman and L. A. Adamic, "Internet: growth dynamics of the world-wide web," *Nature*, vol. 401, no. 6749, pp. 131–131, 1999.

[32] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu, "Graph homomorphism revisited for graph matching," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1161–1172, 2010.

[33] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu, "Graph pattern matching: from intractable to polynomial time," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 264–275, 2010.

[34] S. Ma, Y. Cao, J. Huai, and T. Wo, "Distributed graph pattern matching," in *Proceedings of the 21st international conference on World Wide Web*. ACM, 2012, pp. 949–958.