PARALLEL ALGORITHMS FOR SUBGRAPH PATTERN MATCHING

by

AYUSHI JAIN

(Under the Direction of John A. Miller)

ABSTRACT

Due to the growing importance of Big Data, graphs are becoming huge in size and are rapidly getting too large for conventional computer approaches. Graph Pattern Matching is often defined in terms of subgraph isomorphism, an NP-Complete problem. Most existing graph pattern matching algorithms are very compute intensive. Unfortunately, for such massive graphs, sequential approaches are almost unfeasible. Therefore, parallel computing resources are required to meet their computational and memory requirements. The paper presents a novel parallel subgraph pattern matching algorithm, known as ParDualIso based on Akka. Since, the sequential implementation of ParDualIso known as DualIso adapts Dual Simulation as the pruning technique, so we also present the parallel implementation of Dual Simulation, referred as ParDualSim. The runtimes of the algorithms are tested against their sequential counter-parts on massive graphs of 10 million vertices and 250 million edges.

INDEX WORDS: subgraph isomorphism; parallel algorithms; dual simulation; Akka

PARALLEL ALGORITHMS FOR SUBGRAPH PATTERN MATCHING

by

AYUSHI JAIN

B.Tech., Uttar Pradesh Technical University, INDIA, 2011

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

2014

PARALLEL ALGORITHMS FOR SUBGRAPH PATTERN MATCHING

by

AYUSHI JAIN

Major Professor: John A. Miller

Committee: Lakshmish Ramaswamy
Krzysztof J. Kochut

Electronic Version Approved:

Julie Coffield
Interim Dean of the Graduate School
The University of Georgia
August 2014

DEDICATION

To my friends, family and professors for their endless support, care, belief and motivation.

ACKNOWLEDGEMENTS

Studying in University of Georgia was a great experience for me from past 2 years. I have grown as a better person on both personal and professional front. The knowledge that I gained here is incredible. I learned a lot from my major professor Dr. John A. Miller. He is one of the most intellectual person I have ever met. His knowledge and support has always been the motivating factor for me in completing my research successfully. Also, I would like to extend my gratitude to Dr. Lakshmish Ramaswamy for his endless support and advice. Dr. Krzysztof J. Kochut is one of the best professors and I really enjoyed his Enterprise Integration class. I would also like to give my special thanks to Arash Fard and Matthew Saltz for their constant help and guidance.

.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION AND LITERATURE SURVEY

Graph Analytic has been an important research area over the several years and graph algorithms are becoming very important in so many applications. With this complete new era of Big Data, every other thing is going on-line like social networks, data mining, crowd sourcing, graph databases and so many other domains. Due to the increasingly growing size of data, scalable problems need to be designed. As a result, parallel computing resources are required to meet their computational and memory requirements. Our main topic of interests is Graph Pattern Matching Algorithms [1] which has gained lots of importance due to the growing importance of social networks, genetics, etc. We study several techniques for parallel and distributed implementation of such algorithms and discuss several issues faced while solving these large-scale graph algorithms.

GRAPH PATTERN MATCHING PROBLEM

The objective of all pattern matching algorithms is to find out all the matches of a given graph, called as query graph, in an existing larger graph, called as data graph. Graph based pattern matching is not a single problem, but a set of related problems. These range from exact to inexact matching. Finding an exact match of a query graph in a data graph leads to NP-Complete problem [2] which is known as subgraph isomorphism. In recent years, several variants of graph pattern matching paradigm have been proposed to conduct in polynomial time. They fall into the category of inexact matching. These includes graph

simulation [3], dual simulation [4], strong simulation [4], strict simulation [5] and the most latest tight simulation [6]. Some libraries that implements graph pattern matching related problems include igraph [7], Nauty [8], vflib [9], etc.

To handle such large-scale graphs is very crucial and linear solutions have now become infeasible. Most existing graph pattern matching algorithms are highly computation intensive and do not fit to scale with extremely large graphs that characterize many emerging applications. Due to the resource limitation of single processors in graph processing, parallel computing [10] appears to be the most optimal solution.

Graph problems [11] have some inherent characteristics that makes them poorly matched to current computational problem-solving approaches. Here are few challenges that the graph problems presents:

1. Unstructured problems – The data in the graph problems is highly unstructured and irregular. The partitioning of the data is the major challenge which makes it difficult to parallelize a graph problem. So, every graph problem has a certain limit in scalability due to poorly partitioned data.
2. Computations based on data – The graph problems are highly dictated by data dependencies specifically by the vertex and edge structure of the graph. Thus, computation is highly dependent on the data and as a result, it is difficult to design an efficient parallel algorithms as the structure of the computations is not known in advance.
3. Low locality – Graphs provide an intuitive way to model various entities and relationships between them. Such graphs specifically which are used for data analysis possess highly unstructured and irregular relationships and hence a poor

locality. The performance of the processors is dependent upon the exploitation of the locality. Due to poor locality that these graph algorithms possess, high performance is hard to achieve.

4. High data access to computing ratio – Graph algorithms are highly dependent on the exploration of the structure of the graph than in performing large computations. Thus, there is a high ratio of data access to computation for such applications. As a result, the runtime is highly dominated by the wait for memory fetches.

PARALLEL AND DISTRIBUTED GRAPH PROCESSING MODELS & TECHNIQUES

We discuss how the graph problems, software and parallel hardware are dependent to each other in the current state of the art and various issues that that leads to various challenges in solving large-scale graph problems.

1. Distributed memory machines - The most widespread class of parallel machines are distributed memory computers. These machines typically consists of set of processors and memory connected by some high speed network. Such machines are now quite inexpensive and efficiently used for problems that are trivially parallel. MapReduce [12] despite its popularity for big data computation is unsuitable for supporting iterative graph algorithms. As a result, number of distributed/parallel graph processing systems have been proposed like Pregel [13], its open source implementation Apache Giraph [14], Graphlab [15], Kineograph [16], Trinity [17], GPS [18], Grace [19] and Giraph++ [20]. These are most commonly programmed by explicit message passing. Typically, they are all based upon BSP- based vertex-centric programming model. In this model, the users focus on a local action for each

vertex of the data graph. Its usage of the BSP [21] model makes it free of deadlocks. Moreover, it can also provide very high scalability, and well-suited for distributed implementation.

Data is exchanged between the processors by user-controlled messages, usually with the MPI communication library. This model is very easy to program and has been proved to be useful for many graph algorithms specifically pattern matching algorithms.

By design, message passing programs need not necessarily be bulk synchronous in nature. It also exhibits asynchronous communication in an arbitrary manner. The most common platform is AKKA [22], an actor-based message passing model that provides asynchronous message passing communication in between the actors.

2. Partitioned global address space – It is a parallel programming model. It assumes a global memory address space that is logically partitioned and a portion of it is local to each process or thread. E.g. UPC [23] language. PGAS attempts to combine the advantages of a SPMD programming style for distributed memory systems (as employed by MPI) with the data referencing semantics of shared memory systems. Two programming languages that use this model are Chapel [24] and X10 [25].

   Although, fine-grained programs are easier to write than with MPI, the number of threads of control is fixed in this and generally equal to the number of processors. The inability to use dynamic number of threads is a significant loss to the high performing graph software.

3. Shared memory model - In this programming model, tasks share a common address space, which they read and write to asynchronously. Various mechanisms such as locks / semaphores may be used to control access to the shared memory. Machine memory is physically distributed across networked machines, but appears to the user as a single shared memory (global address space). Generically, this approach is referred to as "virtual shared memory".

4. Multi-threaded Models - In the threads model of parallel programming, a single huge process can have multiple smaller processes concurrently executing their tasks. Each thread has a local data, but they also share the entire resources of the process. A thread's work may best be described as a subroutine within the main program. Any thread can execute any subroutine at the same time as other threads. Threads communicate with each other through global memory (updating address locations). This requires synchronization constructs to ensure that no two threads updates the same global address at same time. The programmer is responsible for determining the parallelism. E.g. POSIX threads and OpenMP.

DESIGN ASPECTS OF PARALLEL GRAPH ALGORITHMS

1. Job Granularity – While designing a parallel graph algorithm, the most important thing is to decide and think properly as to where to introduce parallelism. For example, in All-Pair Shortest Path algorithm, for each vertex, single-source shortest path can be employed to compute the paths in parallel, thereby achieving maximum parallelism.

2. Resource Contention – The key here is to have minimum contention in terms of memory, disk, etc. While designing the parallel graph algorithms, we should take care that multiple processes or threads should not access the same resource simultaneously. Otherwise, system will observe performance degradation.
3. Load Balancing – To achieve maximum parallelism, proper load-balancing among the processors is a major task. Each processor should be assigned tasks in such a way that overall computation is balanced across each processor.

However, high performance graph algorithms have been proven to be difficult to develop. The problems that have simple sequential solutions do not necessarily have a practical parallel solution and may be no efficient parallel solution at all. Various graph frameworks and libraries have been designed which efficiently handle the graph pattern matching problems. These frameworks should be highly flexible, extensible, portable and maintainable. The most common example is Depth First Search, which is often used as a basis of many parallel graph pattern matching algorithms. Graph Pattern Matching also falls into similar category and its efficient parallel and distributed implementation is an open research area.

CHAPTER 2

PARALLEL ALGORITHMS FOR SUBGRAPH PATTERN MATCHING[1]

[1] Ayushi Jain, Aravind Kalimurthy, Matthew Saltz, John A. Miller, to be submitted to International Journal of Big Data 2014.

ABSTRACT

Due to the growing importance of Big Data, graphs are becoming huge in size and are rapidly getting too large for conventional computer approaches. Graph Pattern Matching is often defined in terms of subgraph isomorphism, an NP-Complete problem. Most existing graph pattern matching algorithms are very compute intensive. As a result, parallel and distributed computing may be required to meet their computational and memory requirements. In recent years, many graph processing frameworks have been developed such as Pregel, Giraph, etc. However, they follow vertex centric, Bulk-Synchronous Parallel processing model. Although some recent work such as Giraph++ follows graph-centric BSP approach, our goal is to come up algorithms that reduce synchronization bottlenecks of the BSP model, thereby achieving improved performance. The paper presents a novel parallel subgraph pattern matching algorithm, known as ParDualIso based on an actor-based concurrency asynchronous model. The sequential implementation of ParDualIso known as DualIso adapts Dual Simulation as the pruning technique, so we also present a parallel implementation of Dual Simulation. The runtimes of the algorithms are tested against their sequential counter-parts on massive graphs of size of up to 10 million vertices and 250 million edges. We demonstrate through our experimental results the scalability and performance of the algorithms.

2.1 INRODUCTION

Graph analytics [1] has been an important research area over the past several years and graph algorithms are becoming very important in many applications. With this complete new era of Big Data, many things are going online like social networks [26], data mining [27], crowd sourcing, graph databases [28] and various other domains. Graph pattern matching [29] is an interesting problem in this domain. The problem is given a small query graph, find all its embeddings in a large data graph. Due to the rapidly growing size of data, scalable solutions need to be designed. However, the scalability of these solutions is hindered by the NP-completeness of the subgraph isomorphism problem.

Numerous applications deal with large-scale massive graphs. Graphs are used to represent relationships between the members of a social network, links between the web-pages on the Internet, chemical bonds, etc. [30]. The storage and the analysis of the graph is also an open research area. Efficient query processing [31] on graphs is also becoming popular due to the growing importance of graph databases. The task of the query processing is to find all the subgraphs of the data graph that are similar to the given query graph. This leads to the subgraph isomorphism problem which is NP-complete [64]. This means that exact query matching is intractable in worst case scenarios. Thus, to handle such large-scale graphs and perform exact query matching have become very crucial and sequential solutions may exhibit inadequate performance. Parallel and distributed computing offer solutions to utilize the capabilities of multi-core systems and clusters in graph processing.

Ullmann's algorithm [32] and the VF2 algorithm [33] are pioneering work on subgraph isomorphism and are widely used. They can be effectively used for labeled graphs with thousands of vertices. Apart from these, several state-of-the-art algorithms [34]

have been developed for subgraph isomorphism, including SPath [35], GADDI [36], GraphQL [37], QuickSI [38], SUMMA [39], Turbo-ISO [40] and the very recent DualIso [41]. These are all sequential algorithms that do not exploit the capabilities of multi-core machines or clusters. Two well-known parallel implementations have also been created: Ullmann [42] and STW [43].

Considering the benefits of parallel computing, we use efficient graph exploration and design a parallel version of DualIso [41], an in-memory, pruning-based algorithm for subgraph isomorphism. We call it as ParDualIso throughout this paper. We chose Akka [22] as a programming model for our implementation and discuss the advantages of it in the section 2.2. Since, the high performance of DualIso algorithm is largely due to the effectiveness of the pruning algorithm, known as Dual Simulation [43], we also implemented a parallel version of Dual Simulation and call it as ParDualSim. For our experimentation, we have compared the runtime of our parallel algorithms against their sequential counterparts on both synthetic and real data graphs of up to 10 million vertices and 250 million edges. The performance and scalability of our algorithms is evaluated against several parameters.

The remaining sections of the paper are organized as follows. Section 2.2 presents background information on the subgraph isomorphism problem and the terms used in the rest of the paper. It also includes information about the Akka toolkit and an explanation why we chose this and how suitable it is for our algorithms. Section 2.3 discusses several challenges that the graph pattern matching problems possess to make them parallel. In section 2.4, we explain the parallel implementation of the Dual Simulation algorithm. Section 2.5 is dedicated to the detailed explanation of our parallel subgraph isomorphism

algorithm. Extensive experiments are covered in section 2.6 of the paper to illustrate the behavior of our algorithms. Next, we cover related work in section 2.7 followed by conclusions and future work in section 2.8.

2.2 BACKGROUND

Graphs have always been an important element in the fields of mathematics, computer science, chemistry, biotechnology (among other fields), and have become even more interesting for online social networks such as Facebook, LinkedIn, and Twitter, etc. Graphs are very attractive when it comes to modeling real world data as they are so intuitive and flexible. Consequentially, there are several graph databases available, Neo4j being one of the most popular. However, the problem arises when processing very large graphs comprising of huge number of vertices. There are numerous algorithms for graph processing with immediate applications like subgraph pattern matching [1], maximum common subgraph [48], graph homomorphism [49], isomorphism [50], monomorphism [51], etc.

Graphs can be both directed and undirected depending upon the applications. The vertices and edges of the graph can have certain attributes such as vertex labels, edge labels, etc. Our work is focused only on directed graphs with only vertex labels. Throughout this paper, we have kept the same terminology as in [41] to maintain consistency. A vertex-labeled directed graph is a triple $G(V, E, l)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, and $l: V \rightarrow \boldsymbol{Z}$ assigns a label (an integer) to each vertex. For a given vertex, $v \in V$, we use $adj(v)$ to denote the adjacency set of v such that for any $v \in V$, $adj(v) = \{v': (v, v') \in E\}$. We sometimes refer to the vertices in $adj(v)$ as the children of vertex v, and

conversely, *v* as the parent of all vertices in *adj(v)*. Also, we use the term *degree* to refer to the outdegree of a vertex; i.e., the size of its adjacency set. We also assume the query graph, *Q* to be a connected graph as the disconnected query graph would mean having multiple queries for the same data graph, *G*.

2.2.1 GRAPH PATTERN MATCHING.

The objective of graph pattern matching problems [1] is to find the matches of a given graph, called as query graph *Q*, in an existing larger graph, called as data graph, *G*. The matches are based upon the correspondence between the vertices and edges of two graphs that satisfies more or less stringent constraints [1]. The problem is in huge demand in growing domains of social networking, science, data analysis, query processing on graph databases, and many others. It is typically defined in terms of subgraph isomorphism.

SUBGRAPH ISOMORPHISM. The aim of this problem is to find out all the subgraphs of a data graph that are an exact match to the query graph. This is traditionally the most popular of all the graph pattern matching problems. Below we define the terms *subgraph* and *subgraph isomorphism* mentioned in [41].

Definition: (Subgraph) Given a graph $G(V, E, l)$, a graph $G'(V', E', l')$ is said to be a subgraph of G if $V' \subseteq V$, $E' \subseteq E$, and $\forall v \in V'$, $l'(v) = l(v)$.

Definition: (Subgraph Isomorphism) Given a query graph $Q(V_Q, E_Q, l_Q)$ and a data graph $G(V, E, l)$, a subgraph $G'(V', E', l')$ of G is a subgraph isomorphic match to Q if there exists a bijective function $f : V_Q \rightarrow V'$ such that:

1. $\forall v \in V_Q$, $l_Q(v) = l'(f(v))$.
2. An edge (u, v) is in E_Q if and only if $(f(u), f(v))$ is in E'.

DUAL SIMULATION. It is a pattern matching model which defines a relation from vertices of the query graph to vertices of the data graph with the same label. Dual simulation [44] is a straightforward extension of the simple graph simulation model [1], [52]. The ordered pair consisting of vertices $u \in V_Q$ and $v \in V_G$ is a dual match if it satisfies the following four constraints: label constraint, query graph coverage constraint, child constraint and parent constraint. The label constraint requires that the labels of u and v match. The query graph coverage constraint requires that each $u \in V_Q$ must match at least one vertex in V. The child constraint requires that the set of child labels of u must be a subset of child labels of v. Similarly, the parent constraint requires that the set of parent labels of u must be a subset of parent labels of v.

Definition: Query graph $Q\ (V_Q, E_Q, l_Q)$ matches data graph $G\ (V, E, l)$ via dual simulation, denoted by $Q \trianglelefteq_{sim}^{D} G$, if there is a relation $R_D \subseteq V_Q \times V$ such that:

1) If $(u, v) \in R_D$ then $l_Q(u) = l(v)$
2) $\forall\ u \in V_Q, \exists\ v \in V$ s.t. $(u, v) \in R_D$
3) $\forall\ (u, u') \in E_Q, \exists\ (v,v') \in E$ s.t. $(u, v) \in R_D$ and $(u', v') \in R_D$
4) $\forall\ (u'', u) \in E_Q, \exists\ (v'',v) \in E$ s.t. $(u, v) \in R_D$ and $(u'', v'') \in R_D$

We took the algorithm for dual simulation from [41] which is novel in that it does not require a list of parents for each vertex in the data graph. It uses almost two times less memory than existing algorithms and the runtime is reduced considerably due to the enforcement of both the parent and child constraints inside a single loop. Figure 2.1 shows the data graph and query graph. In the data graph, vertices labeled as ‘d’ are the dual simulation result and ‘i’ represents isomorphic matches.

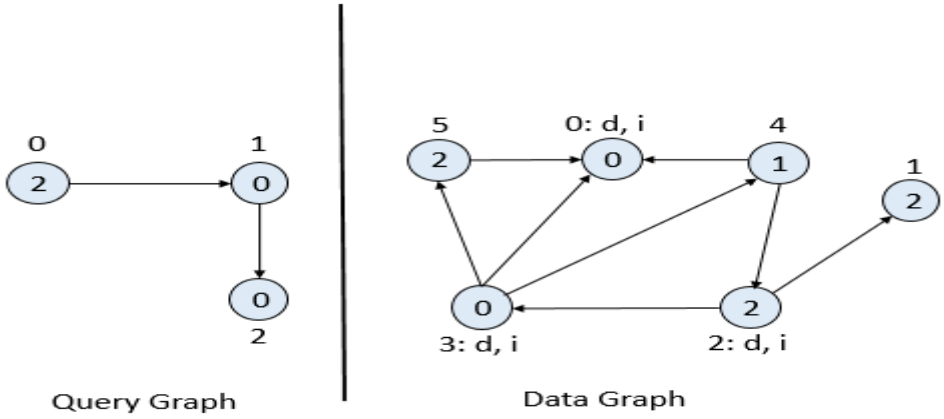

Figure 2.1: Query and Data Graph

2.2.2 MODEL USED.

With “large” graphs comes the desire to extract meaningful information from these graphs. In the age of multi-core CPUs, concurrent processing of graphs proves to be an important topic. The distribution of data among a set of processors has been one of the fundamental issues in parallel processing applications. Undoubtedly, the way one partitions a graph and distributes the vertices among processors affects the performance of the computation. The performance of the system can be measured with respect to two metrics, namely the cost of communication among the processors in the system and the utilization of processors in terms of the time they are not idle. While choosing the framework, the optimal solution is to alleviate the communication cost among the processors and to intensify the utilization of the processors. Further we describe briefly, the model of computation for performing graph algorithms using multiple cores as well as using a cluster of computers. Although the focus of the current work is on parallel processing using multiple cores, the code has been set up to be readily extended to be executed on a cluster.

AKKA. It has been known that building concurrent applications/algorithms is difficult especially when directly managing threads with challenges of handling synchronization.

This is because under-synchronizing the code will lead to race conditions, while over-synchronizing lowers the performance. The actor pattern – Akka [22] makes it easy to separate concerns into isolated actors that allow for safe concurrency. Akka brings the same pattern to Scala.

In an actor-based system, everything is treated as an actor which interact and share information with one another through message passing. There is a layer between the actors and the underlying system with actors processing the messages. The framework handles the thread scheduling and message passing transparently and in a synchronized manner. Some of the key features of this toolkit are high performance with adaptive load-balancing, routing, and remoting based on configurations [54]. With Akka, one can build simple, powerful, concurrent, and distributed applications easily. Also, they are resilient by design which means that the systems built on Akka have a self-healing nature. An actor is a light-weight entity that receives messages and takes actions to handle them. Upon receiving a message, it can process those messages asynchronously using an event-driven receive loop.

Using the Akka framework for performing algorithms on graphs, a feasible way to minimize the communication cost is to partition the graph to a number of highly connected subgraphs with few edges between them and distribute each of these partitions among different processors. We take care of the processor utilization by partitioning the graphs into a number of files that is equal to the number of active processors. Also, we assign each processor their respective partitioned files for which they do the required computations and send back their respective results to the parent/master processor which in turn does the aggregated computation and gives the final results. In our system, there is less communication between the processors, which saves the overall running time for our graph

algorithms to a much greater extent and also saves the communication cost. One of the main reason why we chose this framework is also due to its remarkable feature that allows one to extend the parallel algorithms to distributed ones by making minor changes in its configuration file.

2.2.3 GRAPH PARTITIONING

Graph Partitioning [56] plays a vital role in parallel and distributed graph processing. The problem can be defined on data represented in the form of graph *G(V, E, l)* such that it is possible to partition *G* into smaller components. For instance, a *k-way* partition divides the vertex set into *k* smaller components [56]. We need to first decide the number of partitions based on each experiment we want to perform. There are two things to consider when we do the partitioning. Firstly, a larger partition size increases the chances of the neighboring vertices to belong to the same partition. Secondly, smaller partitions increase the potential degree of parallelism and also help balance the workload across different workers.

However, the more important question is how to partition a graph. A good partitioning strategy should minimize the number of edges crossing different partitions. A wide variety of partitioning and refinement methods can be applied within the overall multi-level scheme. We used METIS components [56] for graph partitioning, which can partition an unstructured graph into *k* parts using either the multilevel recursive bisectioning or the multilevel *k-way* schemes. Both the models can provide high-quality yet different partitions. The algorithms work with a simple goal: edge-cut which basically tries to minimize the edges that travel between the different partitions. Since graph partitioning is not the focus of this work, we leave the in-depth study of the graph

partitioning algorithms to [56]. Referring to [55], we went for the *k-way* partitioning scheme as it was reported to perform better on average than recursive bisectioning. Overall, producing balanced graph partitions with minimum communication cost is an *NP-hard* problem.

2.3 CHALLENGES WITH GRAPH PATTERN MATCHING PROBLEMS

Graph pattern matching problems [11] have some inherent characteristics that makes them poorly matched to current computational problem-solving approaches. The data in the graph problems is highly unstructured and irregular. The partitioning of the data is the major challenge which makes it difficult to parallelize a graph problem. So, every graph problem has a certain limit in scalability due to poorly partitioned data. Efficient data access is not possible due to the poor locality of graphs. The performance of the processors is dependent upon the exploitation of the locality. Due to the poor locality that these graph algorithms possess, high performance is hard to achieve. These graph pattern matching problems are highly dictated by data dependencies specifically by the vertex and edge structure of the graph. Thus, computation is highly dependent on the data and consequently, it is difficult to design an efficient parallel algorithm as the structure of the computations is not known in advance. Moreover, graph algorithms are highly dependent on the exploration of the structure of the graph than in performing large computations. Thus, there is a high ratio of data access to computation for such applications.

In recent years, several platforms have been designed with the aim of harnessing shared nothing clusters for processing massive graphs. MapReduce [12] despite its popularity for big data computation is unsuitable for supporting iterative graph algorithms.

Therefore, a number of distributed/parallel graph processing systems have been proposed like Pregel [13], its open source implementation Apache Giraph [14], GraphLab [15], Kineograph [16], Trinity [17], GPS [18], and GRACE [19]. These are most commonly programmed by explicit message passing. Typically, they are all based upon BSP- based vertex-centric programming model. BSP model may cause synchronization bottlenecks which can be deleterious for load-balancing and maximum parallelism cannot be achieved.

However, while designing the parallel graph based algorithms, there are a few other things that need to be kept in mind, apart from addressing the above mentioned challenges. The most important thing is to decide and think properly as to where to introduce parallelism. For example, in the All-Pair Shortest Path algorithm, for each vertex, single-source shortest path can be employed to compute the paths in parallel, thereby achieving maximum parallelism. Although in general, while designing parallel graph algorithms, one should be careful with multiple processes or threads that access the same resource simultaneously, this is not an issue when using Akka.

We have considered these aspects while implementing our parallel subgraph pattern matching algorithm and have thus achieved considerable performance as can be seen in the experiments section 2.6.

2.4 PARDUALSIM

Since it is the core element of our subgraph isomorphism algorithm, so we now focus on Dual Simulation. In this section, we discuss the implementation of a parallel dual simulation algorithm, referred as *ParDualSim* and later move forward with the discussion of parallel DualIso, referred as *ParDualIso* in the next section.

Below, we highlight an outline for the ParDualSim algorithm that we designed for an actor-based concurrency model:

1. Partition the data graph using METIS.
2. Start Akka ParDualSim System, refer Figure 2.2.
    a. Master assigns task to the workers, number of workers equal to number of partitions.
    b. Each worker performs DualSim4Worker algorithm, independently.
    c. Master aggregates the result.
    d. Master performs dual simulation on the aggregated result.
3. Display dual simulation result.

2.4.1 PARTITIONING APPROACH.

The challenge here is not only to do graph partitioning but also how to maintain boundary vertices information. This is crucial because a poor approach can result into large number of communications, hence performance degradation. The way we perform graph partitioning, induces no communication between the workers for boundary vertices information. Our approach is a four step process:

1. Convert data graph into the format of METIS input graph.
2. Run METIS to generate an output file required for partitioning.
3. Generate the partitioned graphs using data graph and METIS output.
4. Load the partitioned graphs for the workers.

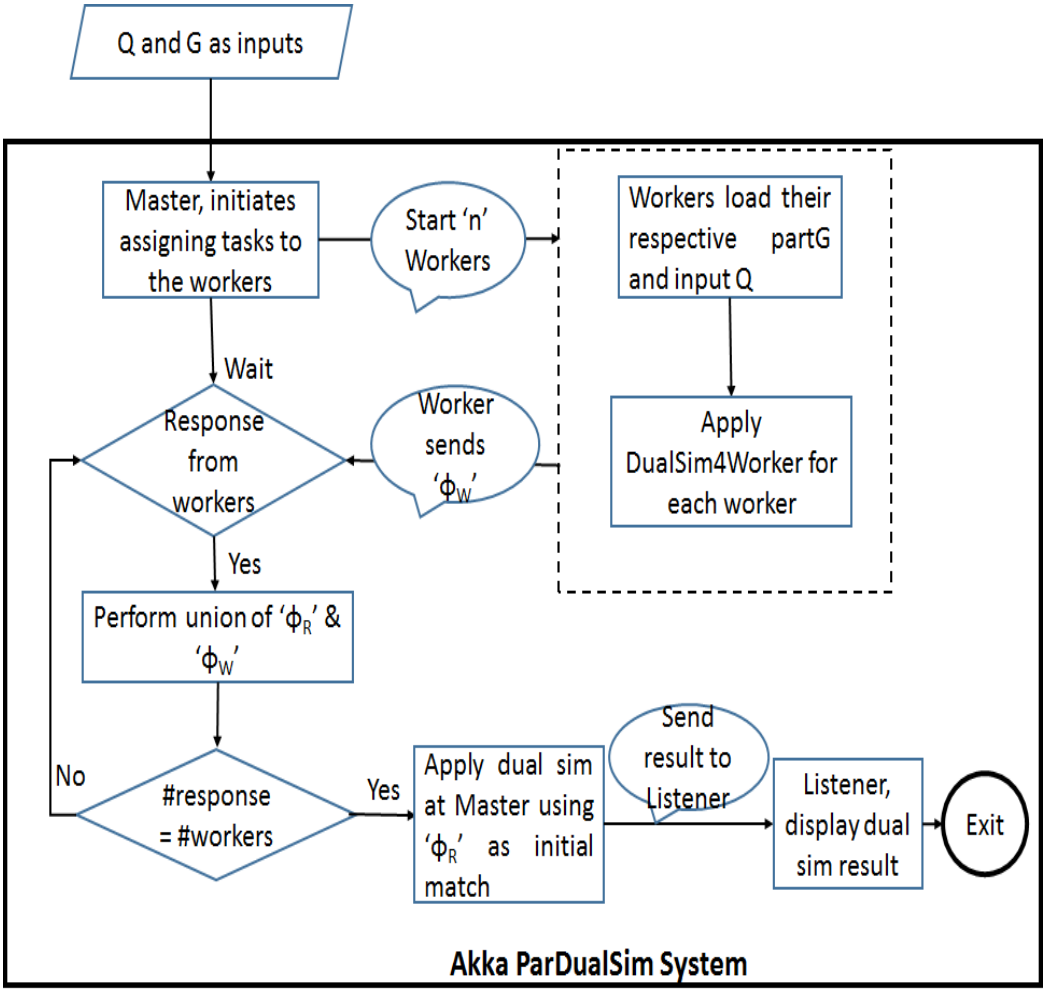

Figure 2.2. The flowchart of Akka ParDualSim system.

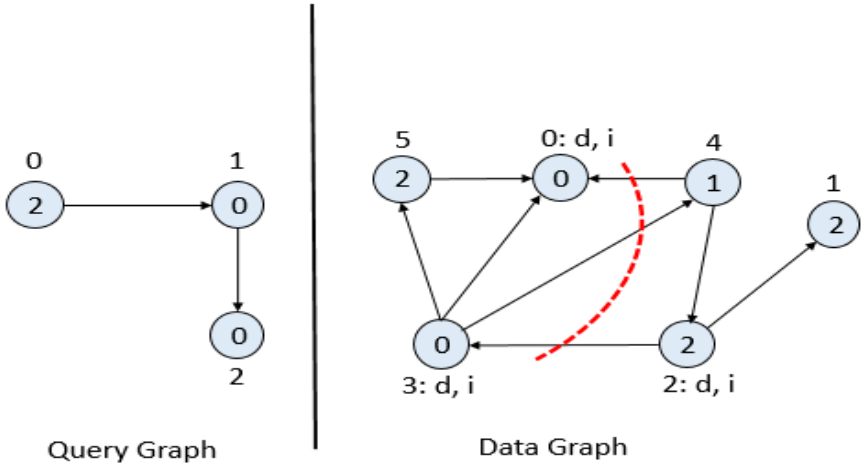

Figure 2.3. The data graph with cuts.

Firstly, all our graphs are directed, while METIS partitions only the undirected graphs. Once we have generated data and query graph randomly through our graph generator, we convert the data graph to properly fit into the format of the METIS. Secondly, after conversion, we run METIS to partition the graph in which we feed the number of partitions that we want for the graph. METIS considers min-edge cut and load-balancing factors while partitioning. For our purpose, we have used the default value of these parameters that METIS provides. The output file includes partition number at each line. This partition number is to map a vertex to its corresponding partitioned graph. For the data graph in Figure 2.3, considering number of partitions equal to 2, the content in the generated output file is:

1, 0, 0, 1, 0, 1

Thirdly, this step is the most crucial step, largely due to the way we generate partitioned graphs and maintain boundary vertices information. Now, we consider the input data graph and the METIS output file together and then partition the graph into the number of partitions in such a way:

a. Generate a number of files equal to the number of partitions where each line of the data graph is copied to the partitioned file number mentioned in the METIS output. The generated partitioned graphs are in the form of:

$partG_i$

vertex id; *label*; {*adjList*}

$partG_0$	$partG_1$
1; 2; {}	0; 0; {}
2; 2; {1, 3}	3; 0; {0, 4, 5}

4; 1; {0, 2} 5; 2; {0}

b. The next task is how to take care of the boundary vertex information. As we know, to perform dual simulation pruning, we need parent and child vertex label information. Considering $partG_0$, vertex 2 has child vertex 3. However, vertex 3 is in $partG_1$. So, while performing dual simulation, we need to communicate in order to get the vertex 3 label information. To reduce this communication, while partitioning we store the parent and child label information separated by break line. Only those vertices will be mentioned below the line which are not the part of actual partitioned graph. For example, we can have information as shown here,

$partG_0$	$partG_1$
1; 2; {}	0; 0; {}
2; 2; {1, 3}	3; 0; {0, 4, 5}
4; 1; {0, 2}	5; 2; {0}
---	---
3; 0;	2; 2;
0; 0;	4; 1;

This approach has an advantage that at the time of loading the partitioned graph itself, the workers can load boundary information also. Therefore, while performing dual simulation, unnecessary communication between the workers is avoided.

2.4.2 GRAPH LOADING FOR WORKER.

Each worker has been assigned a task to perform dual simulation for a partitioned graph. So, a worker loads its corresponding partitioned graph. Workers load the file in such a way

that vertices which are above the break line are the only vertices that belongs to this particular partitioned graph.

So, for $partG_0$ $partG_1$

$adj(1) = \{\}$ $adj(0) = \{\}$

$adj(2) = \{1, 3\}$ $adj(3) = \{0, 4, 5\}$

$adj(4) = \{0, 2\}$ $adj(5) = \{0\}$

However, we construct the label map on the basis of full file information, i.e., including the boundary vertex information too. So, the label map for this example, will be:

$partG_0$ - $\{(1 \rightarrow 2), (2 \rightarrow 2), (4 \rightarrow 1), (3 \rightarrow 0), (0 \rightarrow 0)\}$

$partG_1$ - $\{(0 \rightarrow 0), (3 \rightarrow 0), (5 \rightarrow 2), (2 \rightarrow 2), (4 \rightarrow 1)\}$

2.4.3 DUAL SIMULATION FOR WORKERS.

The key to success for ParDualSim is a modification of the dual simulation [41] algorithm. This modified algorithm is performed by the workers who run the dual simulation for the first iteration only as illustrated in the pseudo-code in Figure 2.4. Our algorithm starts by obtaining feasible matches in the similar way as in [41]. Given a query vertex u, $\Phi(u)$ is created which contains all the vertices of the data graph with the same label as u. Thus, feasible match sets is denoted by Φ. Below, we discuss the steps performed by the workers.

a. The initial feasible matches are calculated on the basis of label map, and since we saw how the workers construct their respective label maps, we will have feasible matches consisting of all the vertices mentioned in the partitioned graph.

   For, $partG_0$ $partG_1$

   $\Phi(0) = \{1, 2\}$ $\Phi(0) = \{2, 5\}$

   $\Phi(1) = \{0, 3\}$ $\Phi(1) = \{0, 3\}$

$\Phi(2) = \{0, 3\}$ $\Phi(2) = \{0, 3\}$

b. Dual simulation is applied using initial feasible match to prune out the unwanted vertices. However, in this case, worker for $partG_0$, does not have capability to prune out vertex 3, as it is not the part of its own partitioned graph. Therefore, worker cannot evaluate those vertices which are the part of boundary information.

c. To deal with this situation, we make two copies of Φ, consisting only of vertices of the partitioned graph. We refer them as Φ_{WC} and Φ_W, respectively.

Initially, $\Phi_W = \Phi_{WC}$.

For, $partG_0$ $partG_1$

$\Phi_W(0) = \{1, 2\}$ $\Phi_W(0) = \{5\}$

$\Phi_W(1) = \{\}$ $\Phi_W(1) = \{0, 3\}$

$\Phi_W(2) = \{\}$ $\Phi_W(2) = \{0, 3\}$

d. Since every worker just needs to evaluate their part's vertices, we iterate over the loop for Φ_{WC}, $v \rightarrow \Phi_{WC}(u)$.

```
1: procedure DUALSIM4WORKER (partGi, Q, Φ, ΦWC, ΦW):
2:     for u ← VQ do
3:         for u′ ← Q.adj(u) do
4:             Φ′(u′) ← ∅
5:             for v ← ΦWC(u) do
6:                 Φv(u′) ← partGi.adj(v) ∩ Φ(u′)
7:                 if Φv(u′) is empty then
8:                     remove v from ΦW(u)
9:                 else add v in ΦW(u)
```

10: $\Phi'(u') \leftarrow \Phi'(u') \cup \Phi v(u')$

11: **if** $\Phi_W(u)$ is not empty **then**

12: $\Phi_W(u') = \Phi'(u')$

13: $\Phi_{WC}(u) = \Phi_W(u)$

14: **return** Φ_W

Figure 2.4. Pseudo code of dual simulation worker.

e. As discussed in section 2.4.2, $partG_i.adj(v)$ may contain some vertices which are not the part of this $partG_i$. According to step c, these vertices will not be in Φ_{WC} and Φ_W. So, the intersection at line 6 of Figure 2.4 can be empty, and as a result, vertex v can be removed from Φ_W. However, the vertex v can still be a potential match because $\Phi(u')$ may contain some vertices of $partG_i.adj(v)$. So, here comes the importance of storing the boundary information within the same file. Thus, having that child and parent label information to construct the initial Φ helps in evaluating vertex v. This is why we do intersection of $partG_i.adj(v)$ and $\Phi(u')$.

   For example, for the $partG_0$ file, and $u = 0$, $u' = 1$, $\Phi_{WC}(u) = \{1, 2\}$, $v = 2$; and $partG_0.adj(v) = \{1, 3\}$.

   So, we see that vertex v is a potential match, however, if we do $partG_0.adj(v) \cap \Phi_{WC}(u')$, v gets eliminated. This is because vertex 3 is not the part of this graph and it is a boundary vertex. However, its information is maintained in Φ. Because we do $partG_0.adj(v) \cap \Phi(u')$, vertex v, i.e., vertex 2 will be considered as a potential match.

f. The reason why we maintain two copies of Φ is that because we want to evaluate each and every vertex of the particular part graph, so we run the loop for Φ_{WC}, however, prune out the vertices from Φ_W.

g. Finally, we return Φ_W as a pruned vertex feasible match set. Below is the pruned feasible match set returned by the workers.

For, $partG_0$ $partG_1$

$\Phi_W(0) = \{2\}$ $\Phi_W(0) = \{5\}$

$\Phi_W(1) = \{3\}$ $\Phi_W(1) = \{3\}$

$\Phi_W(2) = \{\}$ $\Phi_W(2) = \{0\}$

2.4.4 COMPUTATION AT MASTER.

The master is responsible for managing the workers, the worker's response and the result aggregations. The step by step procedure is as follows:

a) Assigns the task to the workers. Master sends actor message to the worker to perform DualSim4Worker (Figure 2.4) for the particular partition graph $partG_i$, $i \leq$ number of partitions.

b) It waits till it gets response from all the workers. Workers response are in the form of feasible match Φ_{Wi}.

For, $partG_0$ $partG_1$

$\Phi_W(0) = \{2\}$ $\Phi_W(0) = \{5\}$

$\Phi_W(1) = \{3\}$ $\Phi_W(1) = \{3\}$

$\Phi_W(2) = \{\}$ $\Phi_W(2) = \{0\}$

c) Master maintains Φ_R, which is $\Phi_R = \Phi_R \cup \Phi_{Wi}, \forall i$. According to the paper [59], union of all the Φ_{Wi} will maintain all the potential matches.

$\Phi_R(0) = \{2, 5\}$

$\Phi_R(1) = \{3\}$

$\Phi_R(2) = \{0\}$

d) Once the master has response from all the workers, it performs dual simulation on Φ_R using the input data graph G and query graph Q.

e) Finally, the master sends the result Φ_R, to the listener, which displays the result and exits the system. In this case, the union result obtained after the first iteration of the dual simulation done by the workers is itself the final dual simulation result.

$$\Phi_R(0) = \{2\}$$

$$\Phi_R(1) = \{3\}$$

$$\Phi_R(2) = \{0\}$$

We would like to emphasize the importance of the first iteration of the dual simulation algorithm. Basically, the first iteration prunes out the maximum unwanted vertices and incurs the majority of the elapsed time of dual simulation. This is due to the concurrent pruning of parent and child vertices through a single loop. Through a detailed study and intensive experiments supporting it, we consider the first iteration as an ideal area to achieve parallelism. Thus, at the worker we just perform the first level of pruning as shown in the pseudo-code. The rest is done by the master itself. This is because the way we have partitioned and loaded the graphs for the workers, they prune out all the unwanted vertices belonging to their own partitioned graphs in the first iteration itself. Consequently, further levels/iterations of dual simulation pruning is not beneficial at the workers. Moreover, when the workers return their initial pruned match sets to the master, the remaining computation is done so quickly that it involves very minute part of the total time. We have shown the first iteration time vs. elapsed time taken by the dual simulation for the real as well as synthetic graphs in the experiments section 2.6, referring to Tables 2.2 and 2.3.

2.5 PARDUALISO

We have considered the DualIso algorithm for parallel implementation on the basis of the paper [41] that shows its performance against the other state-of-the-art algorithms.

DUALISO. Similar to the most of the exact graph pattern matching algorithms, DualIso also falls into the category of Ullmann's inspired tree search based algorithm. Although the general form of these algorithms remains the same, differences in the approaches affect the performance largely. Following the similar paradigm, DualIso follows the below mentioned outline.

1) Retrieve feasible matches $\Phi(u)$ for each vertex $u \in V_Q$ by selecting all vertices v of the data graph such that $l(v) = l_Q(u)$. For example, considering Figure 2.1,

   $\Phi(0) = \{1, 2, 5\}$

   $\Phi(1) = \{0, 3\}$

   $\Phi(2) = \{0, 3\}$

2) Prune the global search space using dual simulation.

   $\Phi(0) = \{2\}$

   $\Phi(1) = \{3\}$

   $\Phi(2) = \{0\}$

3) Traverse the remaining matches in a depth-first manner.
   a. For each traversal, apply dual simulation after isolating the matched vertices of the data graph for the corresponding query graph vertices.
   b. If no isomorphic match exists for current mapping, the algorithm backtracks.
   c. Otherwise, search procedure continues recursively until the maximum depth.

Table 2.1: DualIso Matches

Query Vertices	Match
0	2
1	3
2	0

For more details, please refer to the section IV of [41]. We can see that dual simulation is the core as well as the bottleneck of the DualIso algorithm. In order to make DualIso parallel and efficient, the most important part is to create a parallel implementation of dual simulation. The effectiveness of dual simulation at the global search space is that in 90% of the cases it returns only those vertices that are contained in the subgraph isomorphic match. The only situations we get extra vertices are when there exists a cycle in the query graph or when one vertex in the data graph maps to more than one vertex in the query graph. So we see that adapting dual simulation in DualIso at the global search space is quite effective. We also concluded after extensive experiments that, on an average global search space time of DualIso incurs most of the elapsed time.

METHODOLOGY. The ParDualIso follows the same outline of the DualIso algorithm. Inferring from above, we implemented parallel implementation of Dual Simulation, ParDualSim as discussed in the previous section. We have demonstrated in the experimental section that ParDualSim is much more efficient compared to Dual Simulation, referred as DualSim. The ParDualIso is the re-implementation of DualIso algorithm which adapts ParDualSim as a faster and parallel global search space pruning technique.

In ParDualIso, instead of DualSim, we call ParDualSim to reduce the search space. After this initial pruning step, only 5 to 10 percent of the unwanted vertices are left for further pruning. Therefore, we perform the rest of the procedure at the master. It is also well known that making the recursive DFS parallel do not provide significant speed up [63]. Hence, the traversal of the Depth First Search manner is sequential. Despite of that, we have gained significant speed up compared to DualIso.

Below, we highlight an outline for the ParDualIso algorithm that we designed for an actor-based concurrency model:

1. Partition the graph using METIS.
2. Retrieve initial feasible matches.
3. Master sends a message "start global search space pruning" to the master of Akka ParDualSim system to perform global search space pruning.
4. Master waits till the completion of Akka ParDualSim system.
5. On retrieval of the pruned Φ, traverse remaining matches in depth first search manner.

2.6 EXPERIMENTAL STUDY

In this section, we aim to evaluate our proposed parallel algorithms on the basis of various parameters and illustrate their performance on both synthetic as well as real life data sets. The factors considered are based on the impact of the number of workers, size of the data graph, and size of the query graph, number of distinct labels in the data graph and the density of the data graph. Our study attempts to discover and learn about the various tradeoffs based upon different inputs to the algorithms. Our algorithms are implemented

using the Akka framework, which is an open-source toolkit and is included as the part of the Scala source code library.

We generate random graphs with vertices of uniformly distributed degree. Our synthetic graphs connect each vertex to a random number of other vertices based on a given desired average degree. The number of vertices in the data graph is denoted as $|V_G|$. To describe the relationship between the number of edges and number of vertices in the graph, we use the term alpha, denoted by α, such that $|E| = |V|^{\alpha}$. For our purpose, we have used $\alpha = 1.2$ in general but also show the impact of density by varying the value of α. In all our experiments, we have chosen the number of labels $l = 100$, unless explicitly specified, and for the factor of query size, the number of vertices of query graph is denoted by $|V_Q|$.

All our experiments are run on a machine with two 2GHz Intel Xeon E5-2620 CPUs, each having six hyper-threaded cores for a total of 24 threads, and 128GB of DDR3 RAM. The implementation of DualSim, DualIso, ParDualSim and ParDualIso are written in Scala version 2.10. Both ParDualSim and ParDualIso uses Akka version 2.3.2.

We have used our own graph generator written in Scala to synthesize the large randomly generated graphs. For our queries, we chose a random vertex in the data graph and perform a Breadth-First Search (BFS) until it obtains the desired number of vertices. In this way, there is at least one subgraph isomorphic match in the data graph. The average degree of vertices is also chosen such that $\alpha = 1.2$. All the query graphs are connected graphs. We used amazon-2008 and enwiki-2013 real world datasets (Datasets). The amazon-2008 consists of 735,323 vertices and 5,158,388 edges. It is a symmetric graph that describes the similarity among the books. We generated different numbers of labels for this graph. Other real world dataset is enwiki-2013 which consists of 4,206,785 vertices,

101,355,853 edges and 200 labels. It represents a snapshot of the English part of Wikipedia. Both graphs are tested against BFS queries of size 10 with edges preserving $|E| = |V|^{\alpha}$ where $\alpha = 1.2$.

2.6.1 SIGNIFICANCE OF FIRST ITERATION.

Recalling what was discussed in the section 2.4 regarding the importance of the first iteration of dual simulation algorithm, we did extensive experiments to find out the time taken by the first iteration against the total running time of the algorithm. The experiments are tested against both synthetic and real graphs for around 500 different queries. For real graphs, we randomly generated 200 labels and queries of size 10, whereas synthetic graphs have 100 labels and queries of size 20. The Tables 2.2 and 2.3 show the consistency of the time incurred by the first iteration. Subsequently, making the first iteration parallel helped us to achieve significant speedup. The rest of the experiments shows the performance exhibited by our approach.

Table 2.2: Significance of first iteration for synthetic graphs.

$\|V_G\|$	3e6	5e6	7e6	9e6	10e6
First iteration	901.86	1626.67	2616.10	3304.33	4039.08
Total time	910.61	1667.26	2637.34	3344.31	4083.18

Table 2.3: Significance of first iteration for real graphs.

$\|V_G\|$	amazon-2008	enwiki-2013
First iteration	28.71	282.92
Total time	29.34	289.75

2.6.2 IMPACT OF NUMBER OF WORKERS.

This experiment demonstrates the effect of the number of workers. Recalling the strategy explained in section 2.4.2, the number of workers is equal to the number of partitions of the data graph. This resulted in achieving more parallelism and pruning overall. Looking at Figures 2.5 and 2.6, it can be seen that the speedup increases as the number of workers increases. The maximum speedup is achieved for ParDualIso when number of workers = 20. The Figure 2.6 demonstrates the average case scenario. The rest of the experiments are conducted with number of workers = 20.

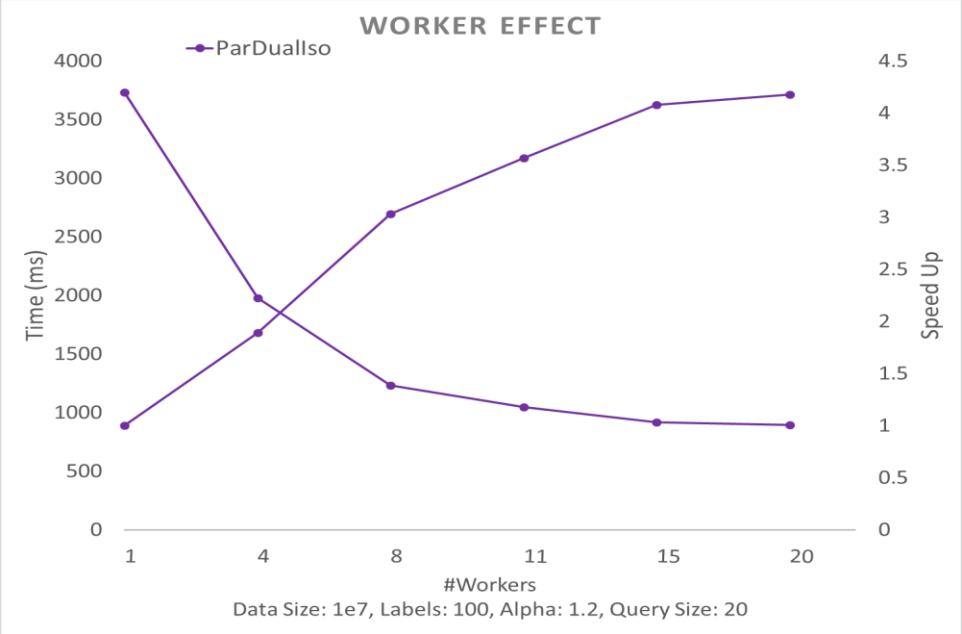

Figure 2.5. The impact of workers on runtime and speed up for ParDualIso.

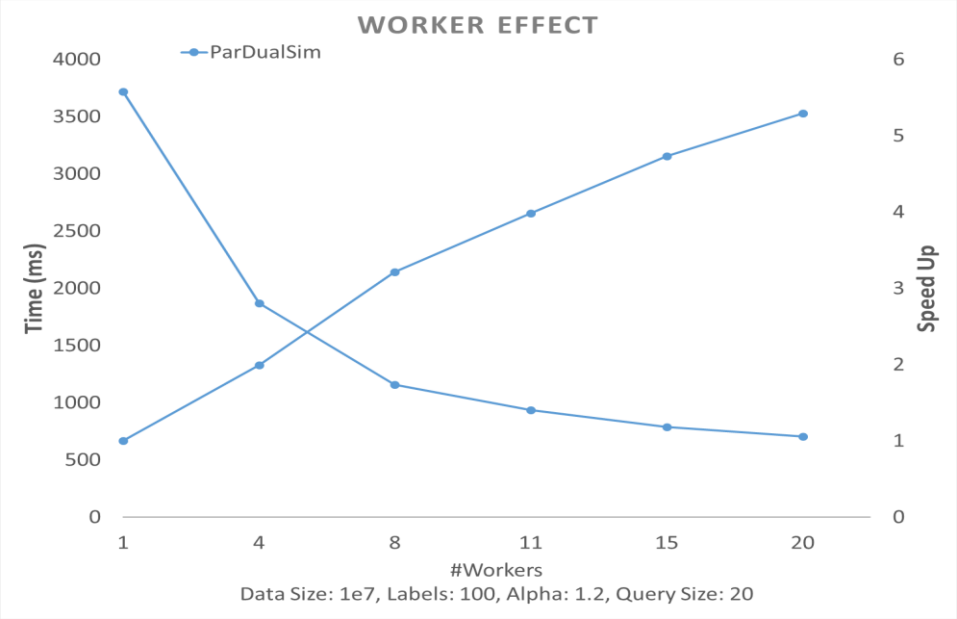

Figure 2.6. The impact of workers on runtime and speed up for ParDualSim.

2.6.3 IMPACT OF DATA GRAPH.

This experiment illustrates the robustness and scalability of the algorithm in terms of the runtime with respect to the size of the data graph denoted by $|V_G|$. The largest data graph is of size 10 million vertices and 250 million edges. As expected the running time of the algorithms increases with the increase in the size of the data graph. The Figures 2.7 and 2.8 show that irrespective of the data graph size, both ParDualIso and ParDualSim achieves an average speed up of 4.5 and 5.5, respectively.

Figure 2.7. The impact of data graph size on runtime for ParDualIso.

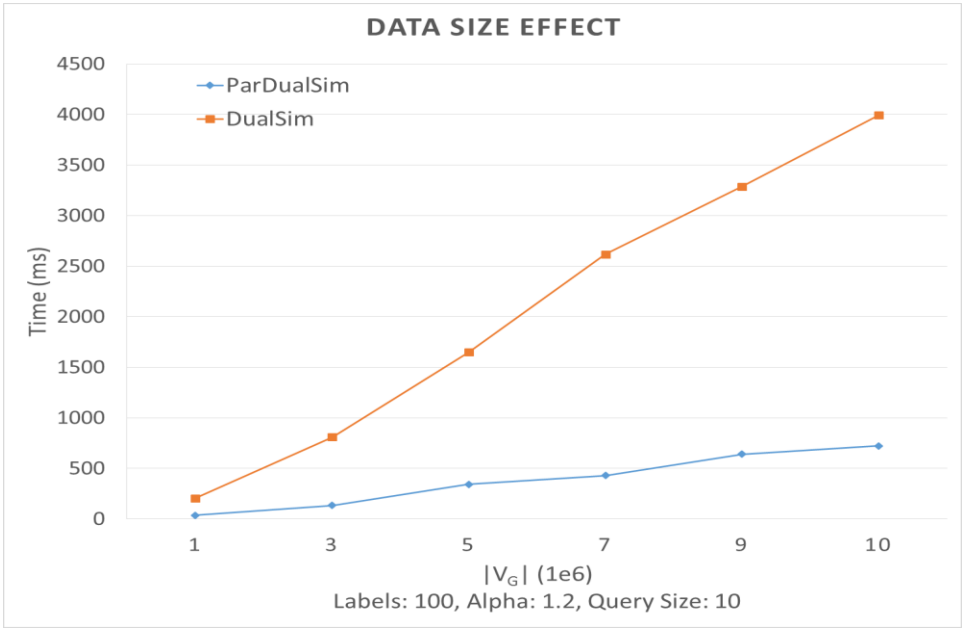

Figure 2.8. The impact of data graph size on runtime for ParDualSim

2.6.4 IMPACT OF QUERY GRAPH.

This experiment demonstrates the impact of different query sizes ranging from 5 to 100 denoted by $|V_Q|$. Again, irrespective of the query size, the ParDualSim is always 4 to 5 times faster than the DualSim. For ParDualIso, we utilize parallelism only for the global search space pruning and the computation of the matches is solely handled by the master. For the queries where the number of matches is large, the master computation time is very high as we do not stop after the first thousand matches as is typical in other algorithms (our limit is one million). Still, ParDualIso gains a speed up of 2 to 5. In the worst scenario, the runtime is approximately the same as of DualIso. The overall runtime against query size mostly depends on the number of matches.

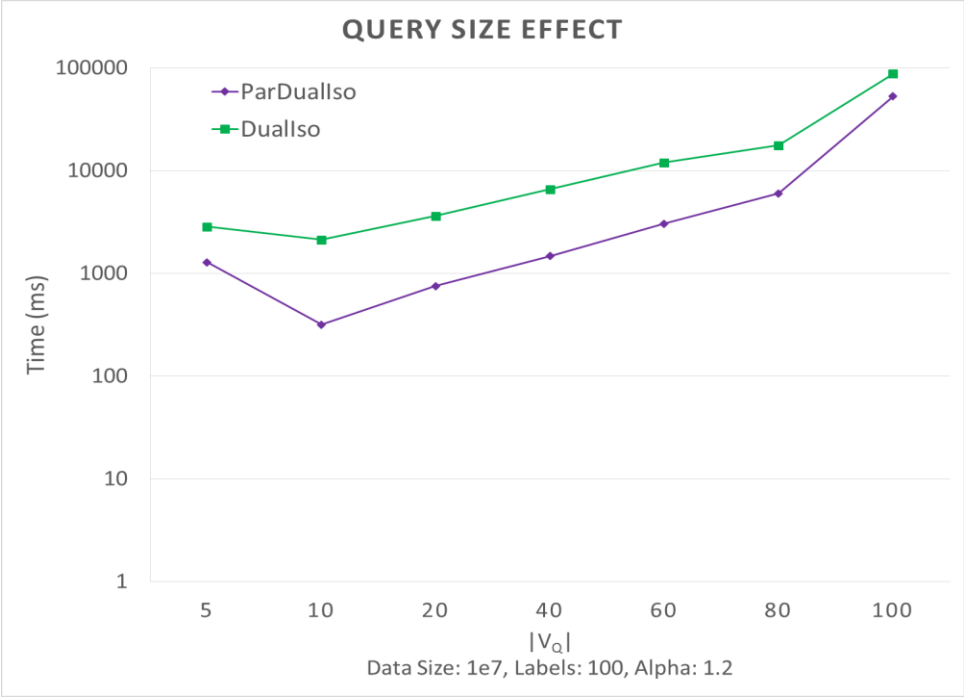

Figure 2.9. The impact of query graph size on runtime for ParDualIso.

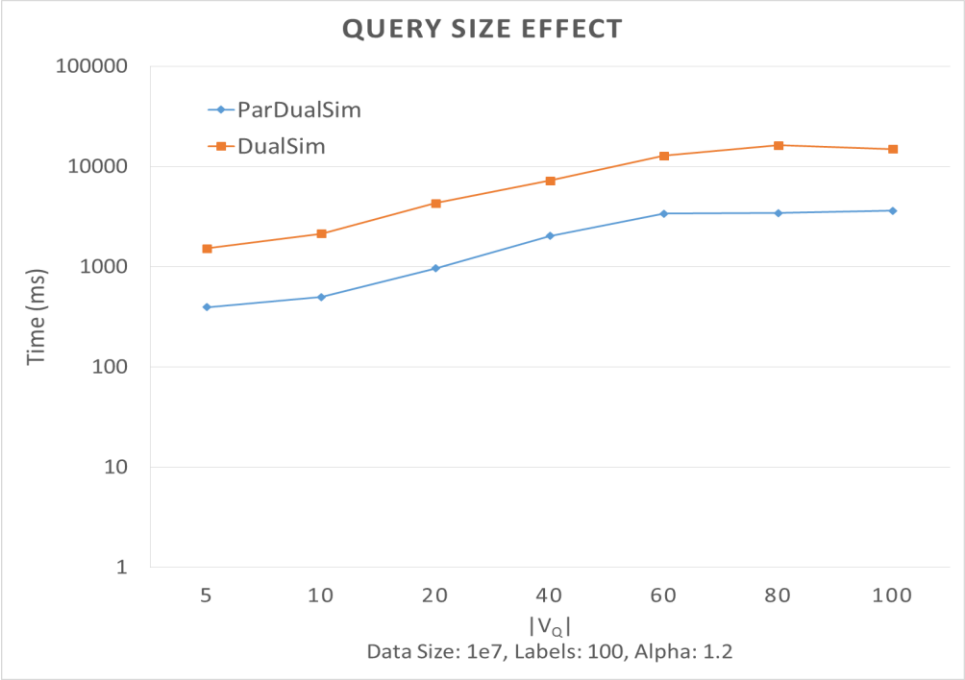

Figure 2.10. The impact of query graph size on runtime for ParDualSim.

2.6.5 IMPACT OF LABELS.

As the feasible matches are constructed on the basis of labels, the more the number of unique labels in the graph, higher the pruning of global search space. This results in less computation at the master and fewer matches comparatively.

The Figures 2.11 and 2.12 illustrate the impact of number of unique labels on the runtime of the algorithms. The speed up of ParDualIso gains proportionally with respect to the number of unique labels.

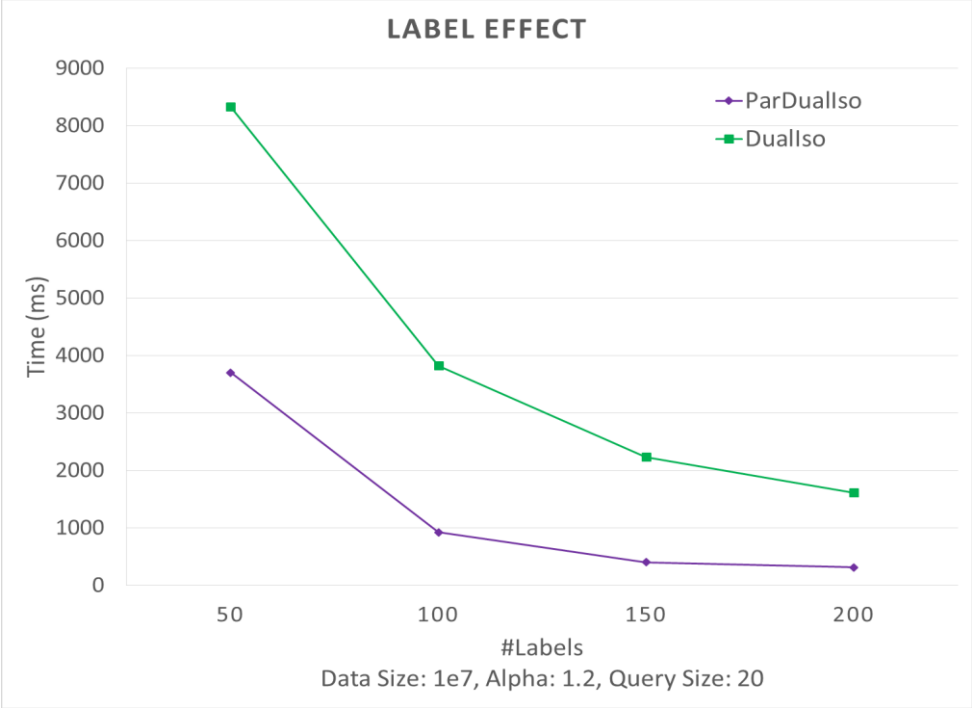

Figure 2.11. The impact of labels on runtime for ParDualIso.

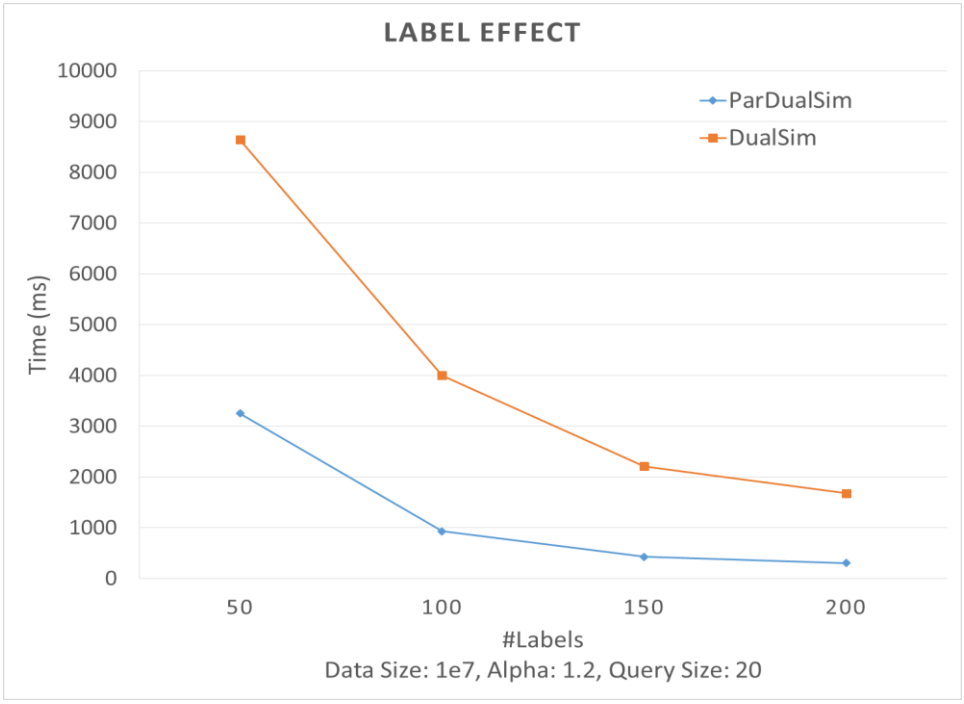

Figure 2.12. The impact of labels on runtime for ParDualSim.

2.6.6 IMPACT OF DENSITY.

The density of the data graph is directly proportional to the number of matches. As the density increases, the speed up of ParDualIso decreases. We gained a speed up of approximately 7.5 when α = 1.1. The Figures 2.13 and 2.14 show the expected behavior of the algorithms.

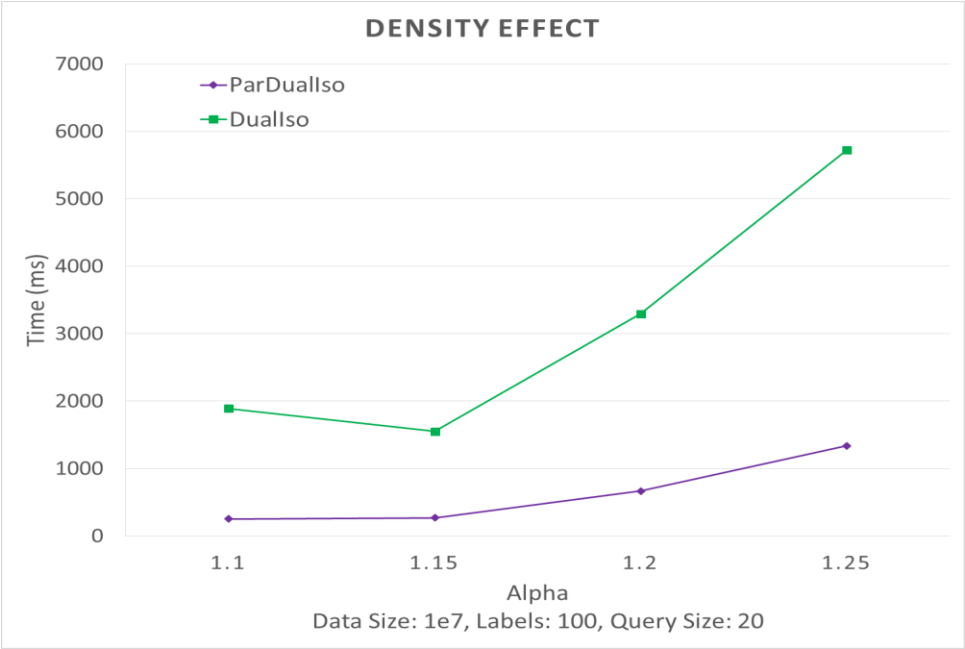

Figure 2.13. The impact of density on runtime for ParDualIso.

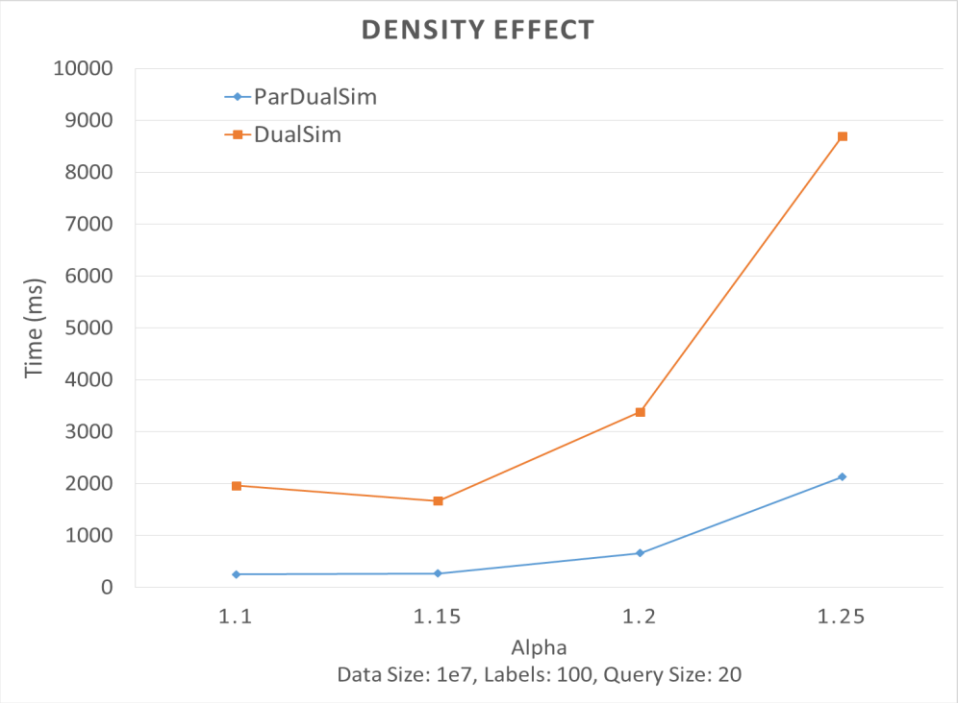

Figure 2.14. The impact of density on runtime for ParDualSim.

2.6.7 IMPACT OF REAL GRAPHS.

Here we demonstrate the performance of our algorithms on the real world graphs. We took two real graphs, *amazon-2008* and *enwiki-2013* as mentioned before.

For *amazon-2008*, we randomly assigned 10, 50 and 200 labels respectively keeping the query size of 20. This data graph has less density. The number of matches obtained for the queries that we generated are less. So, as the number of unique labels increases, the speed up of ParDualIso also increases with respect to DualIso.

However, for *enwiki-2013* data sets, which has 200 labels randomly generated, we obtained a large number of matches for our queries. Sometimes, the number of matches were more than 1 million. Looking at Figures 2.17 and 2.18, we can see that despite of speed up gained by ParDualSim, ParDualIso elapsed time is approximately equal to DualIso. This shows

one of the worst-case scenarios for our algorithm, still ParDualIso will always be at least better than DualIso.

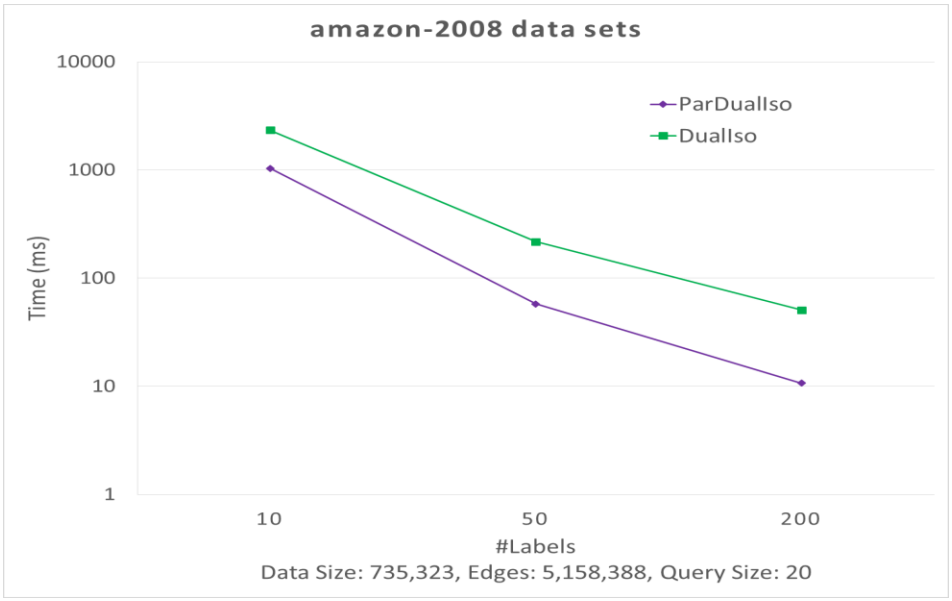

Figure 2.15. The impact of amazon data on runtime for ParDualIso.

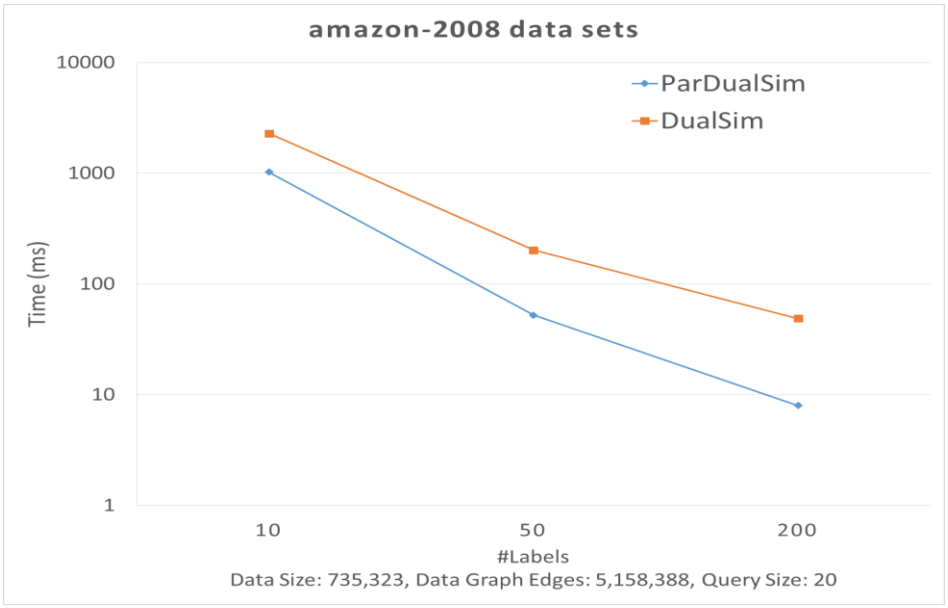

Figure 2.16. The impact of amazon data on runtime for ParDualSim.

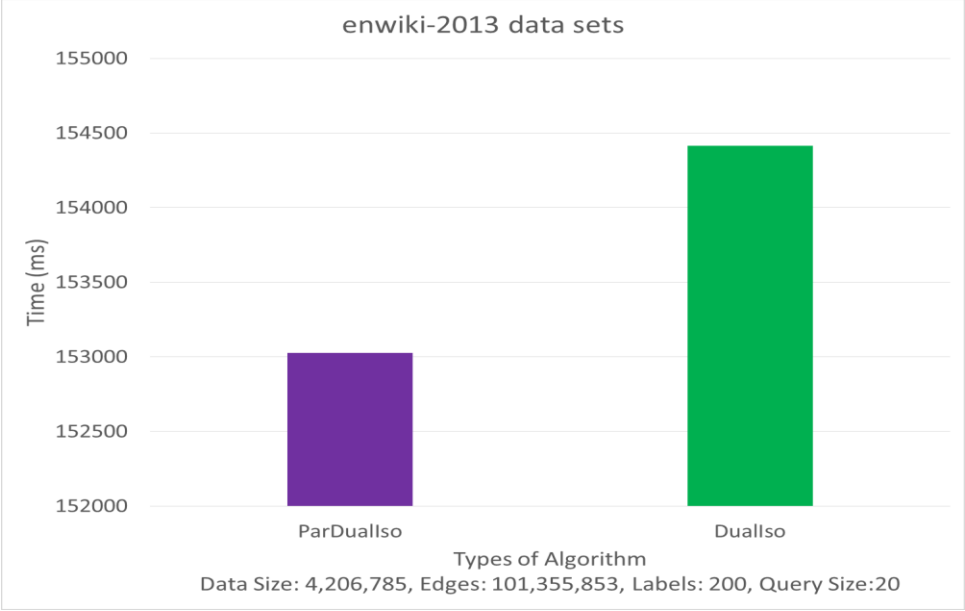

Figure 2.17. The impact of enwiki data on runtime for ParDualIso.

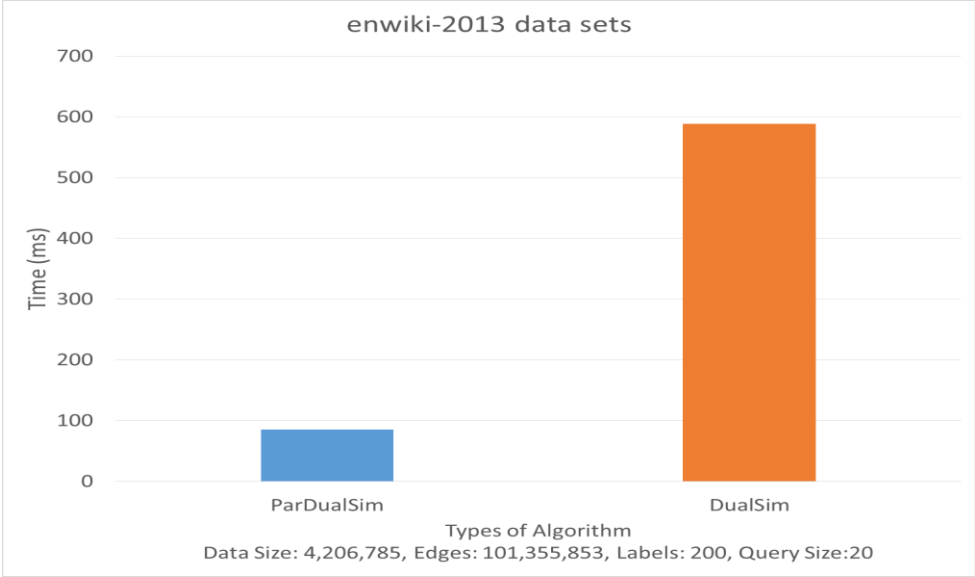

Figure 2.18. The impact of enwiki data on runtime for ParDualSim.

2.7 RELATED WORK

DualIso was introduced in [41]. This paper extends [41] by including 1) Parallel implementation of DualIso, referred to ParDualIso. 2) Parallel implementation of Dual

Simulation, referred to as ParDualSim. 3) A more extensive experimental study compared to the study of [41]. Since graph pattern matching has been widely popular, there has been much research going on in this particular area. We have focused ourselves mainly in parallel and distributed graph pattern matching techniques.

The Ullmann [42] algorithm is the first major algorithm for subgraph isomorphism algorithm, its parallel solution has been implemented on the distributed array processors. Parallelism is achieved using a massive parallel computer, the Active Memory Technology (AMT) and Distributed Array Processor (DAP).

In [43], they presented efficient subgraph matching for graphs deployed on a distributed memory store, Trinity [17]. They used graph exploration but could not avoid expensive join operations because of the limits of subgraph isomorphism. Their focus is on efficient web-scale graph processing, that avoids exhaustive usage of indices which is very common in most of graph processing systems. Since they only use a simple string index which maps node labels to node IDs, the time needed for constructing indices and the capacity for their storage would be infeasible in web-scale graph processing. Moreover, their algorithm gets terminated after obtaining 1024 matches.

In [59], the authors proposed algorithms and optimizations technique for graph simulation in a distributed setting by using a message passing model. They have used a variety of complexity measures on the performance of their algorithms. The algorithms are implemented in Python and tested on a cluster of 16 machines.

Berry, Jonathan W., et al. have introduced the MultiThreaded Graph Library (MTGL) [61]. This is a generic graph query software for processing semantic graphs on multithreaded computers. They have presented a new heuristic for inexact subgraph

isomorphism and have explored the performance and other basic graph algorithms on large scale-free graphs.

In [62], the authors proposed a new pattern matching technique called as strong simulation. They presented the distributed algorithm for strong simulation based on a message-passing model. They consider the graphs to be already partitioned which can have a significant impact on the evaluation of strong simulation. However, ball-creation is a time-consuming process and slight imbalances can overload some machines, thus negatively effecting the overall speed-up.

In [6], they implemented vertex-centric distributed algorithms for simulation models on GPS following BSP approach along with newly proposed tight simulation model. Tight simulation is a novel improvement of strict simulation which was introduced in [5].

Tian, Yuanyuan et al. implemented a new system, called Giraph++ in [20], based on Apache Giraph, an open source implementation of Pregel [13]. They have explored three different categories of graph algorithms, graph traversal, random walk, and graph aggregation, and demonstrate their performance and flexibility on well-partitioned data on this graph-centric model. However, they have not yet explored subgraph pattern matching.

2.8. CONCLUSIONS AND FUTURE WORK

In summary, we have presented a parallel algorithm for subgraph pattern matching, ParDualIso which is a modification of the sequential DualIso algorithm. Since, the effectiveness of DualIso [41] is largely due to the pruning technique adapted, known as Dual Simulation. We also present a parallel algorithm for Dual Simulation, referred as

ParDualSim. Both the algorithms are implemented on Akka. We found that both ParDualIso and ParDualSim achieve significant speedup through the experiments on both synthetic and real graphs.

Future work includes the distributed implementation of these algorithms that could allow the algorithm to handle even larger graphs. We also foresee that parallel implementation of Tight Simulation [6] will also be a good choice for pruning.

CHAPTER 3

SUMMARY

In summary, we have presented a parallel algorithm for subgraph pattern matching, ParDualIso which is a modification of the sequential DualIso algorithm. Since, the effectiveness of DualIso [41] is largely due to the pruning technique adapted, known as Dual Simulation. We also present a parallel algorithm for Dual Simulation, referred as ParDualSim. Both the algorithms are implemented on Akka. We found that both ParDualIso and ParDualSim achieve significant speedup through the experiments on both synthetic and real graphs.

Future work includes the distributed implementation of these algorithms that could allow the algorithm to handle even larger graphs. We also foresee that parallel implementation of Tight Simulation [6] will also be a good choice for pruning.

REFERENCES

[1] Conte, Donatello, et al. "Thirty years of graph matching in pattern recognition." International journal of pattern recognition and artificial intelligence 18.03 (2004): 265-298.

[2] http://en.wikipedia.org/wiki/NP-complete

[3] Milner, Robin. Communication and concurrency. Prentice-Hall, Inc., 1989.

[4] Ma, Shuai, et al. "Capturing topology in graph pattern matching." Proceedings of the VLDB Endowment 4 (2011): 310-321.

[5] Fard, Arash, et al. "A distributed vertex-centric approach for pattern matching in massive graphs." Big Data, 2013 IEEE International Conference on. IEEE, 2013.

[6] Fard, Arash, et al. "DISTRIBUTED AND SCALABLE GRAPH PATTERN MATCHING: MODELS AND ALGORITHMS."

[7] Csardi, Gabor, and Tamas Nepusz. "The igraph software package for complex network research." InterJournal, Complex Systems 1695.5 (2006).

[8] McKay, Brendan D. "Nauty user's guide (version 2.4)." Computer Science Dept., Australian National University (2007).

[9] Foggia, Pasquale. "The vflib graph matching library, version 2.0." (2001).

[10] http://hunch.net/~large_scale_survey/SUML.pdf

[11] Lumsdaine, Andrew, et al. "Challenges in parallel graph processing." Parallel Processing Letters 17.01 (2007): 5-20.

[12] Chu, Cheng, et al. "Map-reduce for machine learning on multicore." Advances in neural information processing systems 19 (2007): 281.

[13] Malewicz, Grzegorz, et al. "Pregel: a system for large-scale graph processing." Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010.

[14] Avery, Ching. "Giraph: Large-scale graph processing infrastruction on Hadoop." Proceedings of Hadoop Summit. Santa Clara, USA:[sn] (2011).

[15] Low, Yucheng, et al. "Graphlab: A new framework for parallel machine learning." arXiv preprint arXiv:1006.4990 (2010).

[16] Cheng, Raymond, et al. "Kineograph: taking the pulse of a fast-changing and connected world." Proceedings of the 7th ACM european conference on Computer Systems. ACM, 2012.

[17] Shao, Bin, Haixun Wang, and Yatao Li. "Trinity: A distributed graph engine on a memory cloud." Proceedings of the 2013 international conference on Management of data. ACM, 2013.

[18] Salihoglu, Semih, and Jennifer Widom. "Gps: A graph processing system." Proceedings of the 25th International Conference on Scientific and Statistical Database Management. ACM, 2013.

[19] Kreowski, Hans-Jörg, and Sabine Kuske. "On the interleaving semantics of transformation units—a step into GRACE." Graph Grammars and Their Application to Computer Science. Springer Berlin Heidelberg, 1996.

[20] Tian, Yuanyuan, et al. "From “think like a vertex” to “think like a graph”." Proceedings of the VLDB Endowment 7.3 (2013).

[21] http://en.wikipedia.org/wiki/Bulk_synchronous_parallel

[22] Imam, S. M., & Sarkar, V. (2012, October). Integrating task parallelism with actors. In ACM SIGPLAN Notices (Vol. 47, No. 10, pp. 753-772). ACM.

[23] https://upc-lang.org/

[24] http://en.wikipedia.org/wiki/Chapel_%28programming_language%29

[25] http://en.wikipedia.org/wiki/X10_%28programming_language%29

[26] Fan, W. (2012, March). Graph pattern matching revised for social network analysis. In Proceedings of the 15th International Conference on Database Theory (pp. 8-21). ACM.

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

[28] Jin, W., & Yang, J. (2011, January). A flexible graph pattern matching framework via indexing. In Scientific and Statistical Database Management (pp. 293-311). Springer Berlin Heidelberg.

[29] Gallagher, B. (2006). Matching structure and semantics: A survey on graph-based pattern matching. AAAI FS, 6, 45-53.

[30] Wang, H. (2010). Managing and mining graph data (Vol. 40). C. C. Aggarwal (Ed.). New York: Springer.

[31] Fard, A., Abdolrashidi, A., Ramaswamy, L., & Miller, J. A. (2012, October). Towards efficient query processing on massive time-evolving graphs. In CollaborateCom (pp. 567-574).

[32] Ullmann, J. R. (1976). An algorithm for subgraph isomorphism. Journal of the ACM (JACM), 23(1), 31-42.

[33] Cordella, L. P., Foggia, P., Sansone, C., & Vento, M. (2004). A (sub) graph isomorphism algorithm for matching large graphs. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 26(10), 1367-1372.

[34] Lee, J., Han, W. S., Kasperovics, R., & Lee, J. H. (2012, December). An in-depth comparison of subgraph isomorphism algorithms in graph databases. In Proceedings of the VLDB Endowment (Vol. 6, No. 2, pp. 133-144). VLDB Endowment.

[35] Zhao, P., & Han, J. (2010). On graph query optimization in large networks. Proceedings of the VLDB Endowment, 3(1-2), 340-351.

[36] Zhang, S., Li, S., & Yang, J. (2009, March). GADDI: distance index based subgraph matching in biological networks. In Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (pp. 192-203). ACM.

[37] He, H., & Singh, A. K. (2008, June). Graphs-at-a-time: query language and access methods for graph databases. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data (pp. 405-418). ACM.

[38] Shang, H., Zhang, Y., Lin, X., & Yu, J. X. (2008). Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. Proceedings of the VLDB Endowment, 1(1), 364-375.

[39] Zhang, S., Li, S., & Yang, J. (2010, October). SUMMA: subgraph matching in massive graphs. In Proceedings of the 19th ACM international conference on Information and knowledge management (pp. 1285-1288). ACM.

[40] Han, W. S., Lee, J., & Lee, J. H. (2013, June). Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In Proceedings of the 2013 international conference on Management of data (pp. 337-348). ACM.

[41] Matthew Saltz, Ayushi Jain, Abhishek Kothari, Arash Fard, John A. Miller, and Lakshmish Ramaswamy. (2014, June). DualIso: A Scalable Subgraph Pattern Matching On Large Labeled Graphs, Proceedings of the 3rd IEEE International Conference on Big Data.

[42] Willett, P., Wilson, T., & Reddaway, S. F. (1991). Atom-by-atom searching using massive parallelism. Implementation of the Ullmann subgraph isomorphism algorithm on the distributed array processor. Journal of chemical information and computer sciences, 31(2), 225-233.

[43] Sun, Z., Wang, H., Wang, H., Shao, B., & Li, J. (2012). Efficient subgraph matching on billion node graphs. Proceedings of the VLDB Endowment, 5(9), 788-799.

[44] Ma, S., Cao, Y., Fan, W., Huai, J., & Wo, T. (2011). Capturing topology in graph pattern matching. Proceedings of the VLDB Endowment, 5(4), 310-321.

[45] Facebook Wikipedia, http://en.wikipedia.org/wiki/Facebook

[46] Twitter Wikipedia, http://en.wikipedia.org/wiki/Twitter

[47] Neo4j graph database, http://www.neo4j.org/

[48] Raymond, J. W., & Willett, P. (2002). Maximum common subgraph isomorphism algorithms for the matching of chemical structures. Journal of computer-aided molecular design, 16(7), 521-533.

[49] Fan, W., Li, J., Ma, S., Wang, H., & Wu, Y. (2010). Graph homomorphism revisited for graph matching. Proceedings of the VLDB Endowment, 3(1-2), 1161-1172.

[50] Corneil, D. G., & Gotlieb, C. C. (1970). An efficient algorithm for graph isomorphism. Journal of the ACM (JACM), 17(1), 51-64.

[51] Wong, A. K., You, M., & Chan, S. C. (1990). An algorithm for graph optimal monomorphism. Systems, Man and Cybernetics, IEEE Transactions on, 20(3), 628-638.

[52] Henzinger, M. R., Henzinger, T. A., & Kopke, P. W. (1995, October). Computing simulations on finite and infinite graphs. In Foundations of Computer Science, 1995. Proceedings, 36th Annual Symposium on (pp. 453-462). IEEE.

[53] Erlang, a programming language used to build massively scalable soft real-time systems, http://www.erlang.org/

[54] Tasharofi, S., Dinges, P., & Johnson, R. E. (2013). Why do scala developers mix the actor model with other concurrency models? In ECOOP 2013–Object-Oriented Programming (pp. 302-326). Springer Berlin Heidelberg.

[55] Nisar, M. U., Fard, A., & Miller, J. A. (2013, June). Techniques for graph analytics on big data. In Big Data (BigData Congress), 2013 IEEE International Congress on (pp. 25-262). IEEE.

[56] Karypis, G., & Kumar, V. (1995). METIS-unstructured graph partitioning and sparse matrix ordering system, version 2.0.

[57] Min edge cut problem, http://en.wikipedia.org/wiki/Minimum_cut

[58] Dreyfus, S. E. (1969). An appraisal of some shortest-path algorithms. Operations research, 17(3), 395-412.

[59] Ma, S., Cao, Y., Huai, J., & Wo, T. (2012, April). Distributed graph pattern matching. In Proceedings of the 21st international conference on World Wide Web (pp. 949-958). ACM.

[60] Boldi, P., & Vigna, S. (2004). The WebGraph Framework: Compression Techniques. Proc. of the Thirteenth International World Wide Web Conference (WWW 2004) (pp. 595--601). Manhattan, USA: ACM Press.

Datasets. (n.d.). Retrieved from Laboratory for Web Algorithmics: http://law.di.unimi.it/datasets.php

[61] Barrett, B. W., Berry, J. W., Murphy, R. C., & Wheeler, K. B. (2009, May). Implementing a portable multi-threaded graph library: The MTGL on Qthreads. In Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on (pp. 1-8). IEEE.

[62] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo, 2014, Strong Simulation: Capturing Topology in Graph Pattern Matching. ACM Trans. Embedd. Comput. Syst.V, N, Article A (January YYYY), 45 pages.

[63] Reif, J. H. (1985). Depth-first search is inherently sequential. Information Processing Letters, 20(5), 229-234.

[64] Cook, Stephen A. "The complexity of theorem-proving procedures." Proceedings of the third annual ACM symposium on Theory of computing. ACM, 1971.