Parallel Subgraph Isomorphism

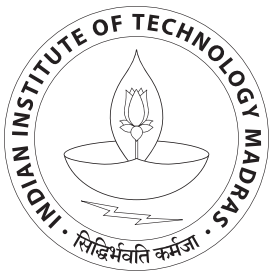
A Project Report

submitted by

SHREYAS PHANSE

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

April 2018

THESIS CERTIFICATE

This is to certify that the thesis entitled **Parallel Subgraph Isomorphism**, submitted by **Shreyas Phanse**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bonafide record of the research work carried out by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Rupesh Nasre
Project Guide
Asst. Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 23 April 2018

ACKNOWLEDGEMENTS

My foremost gratitude and respects to my guide Dr. Rupesh Nasre for his guidance, motivation and moral support in my execution of this project work. His constant motivation for me towards the project helped me throughout the project.

I also thank my friends Mahesh Magar, Maniraja Pedaprolu, Ganesh k. who supported me throughout the project and constantly encouraged me to work harder. I thank my parents and my sister to help my balance my work my constantly motivating me.

ABSTRACT

Subgraph isomorphism is used in many real world problems such as chemical compounds, social networks, and biological structures. Many real applications in bio-informatics, chemistry, and software engineering require efficient and effective management of graph structured data.

One of most important graph queries in graph databases is the subgraph isomorphism query. That is, given a query q and a data graph g, find all embeddings (matches) of q in g. This problem is NP-hard. Various popular algorithms that have tried to solve the problem in reasonable amount of time.

With the development of GPUs and the number of cores, there has been a huge increment in amount of parallelism achieved and therefore increasing the speed up and performance of algorithm. In this project, we have tried to parallelise subgraph isomorphism algorithm and compare it with the sequential implementation. We have studied the scenarios where each of these implementations outperform each other

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

A large number of real world scenarios can be modelled as graphs and a number of real world problems can be solved by using graph algorithms. For example, shortest distance between two cities can be found using the Dijkstras shortest path algorithm by modelling the cities and the roads connecting them as graphs. Similarly, other problems such as relation between people, chemical compounds, DNA sequences etc. can be modelled as graphs and many complex problems related to them can be solved using various graph algorithms.

One of the most important type of graph problem is Subgraph isomorphism. In real world problems which are modelled as graphs, subgraph isomorphism is an important problem. Since this problem is NP-hard, it is not an easy task to find solutions for large graphs in a small amount of time. however, Recent works have been able to solve the problem in quite a reasonable amount of time on specific real-world datasets. Because of numerous applications of the problem in many data mining and pattern matching fields, the problem has been studied through years and different methods have been proposed over time. Some of the state-of-the-art techniques for the same have been studied and discussed in upcoming sections. In recent years, with the development of parallel programming and GPUs, massive speedups have been achieved on execution of various computing

tasks. Our objective is to exploit this to obtain even faster query outputs for large graphs.

1.2 Problem Statement

Graph:

A graph g is defined as a 3-tuple (V, E, L) where V is the set of vertices, $E(\subseteq V \times V)$ is the set of edges, and L is a label function that maps a vertex to a set of labels. Without loss of generality, all subgraph isomorphism algorithms can be easily extended to handle graphs whose edges have labels.

Subgraph Isomorphism:

A graph $q = (V, E, L)$ is isomorphic to a subgraph of a data graph $g = (V', E', L')$ if there is an injection (or an embedding) $M : V \rightarrow V$ such that, $\forall u \in V$, $L(u) \subseteq L'(M(u))$, and $\forall (u_i, u_j) \in E$, $(M(u_i), M(u_j)) \in E'$. Such a subgraph is also called an embedding.

Now, the subgraph isomorphism problem is defined as follows: Given a query graph q and a data graph g, find all distinct embeddings of q in g. This is explained by the following example.

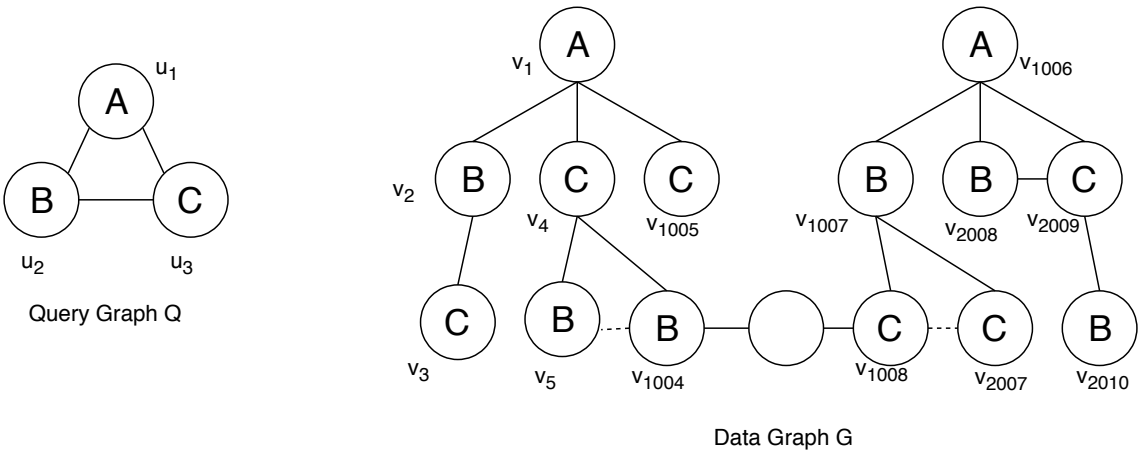


Figure 1.1: A query graph q and a data graph g

For the above query graph, the data subgraph $v_{1006}, v_{2008}, v_{2009}$ is isomorphic to the query graph and hence is a match. There is no other match for the given query graph in the data graph.

1.3 Data Representation

In the implementation, graphs are represented as compressed sparse rows(CSR). The CSR representation of a matrix consists of several one dimensional arrays. The number of arrays depends upon the information carried by the matrix. For example, consider the adjacency matrix of a labelled graph with 9 vertices as shown in the figure below. For this graph, the CSR representation will contain 3 vectors (1-d arrays), namely IA, JA and LA, which are defined as:

IA is an array of size rows(M) + 1 which is recursively defined as:

- $IA[0] = 0$
- $IA[i] = IA[i-1]$ + number of non zero entries in $i - 1^{th}$ row of M

JA is an array of size equal to value last element of IA i.e. the number of edges in the graph. The entries of JA are the column numbers of all the non zero entries of from left to right row-wise.

LA is an array of size rows(M). It contains the labels of all the nodes in order. For the graph in figure 1.2, the matrix M and CSR representation are given.

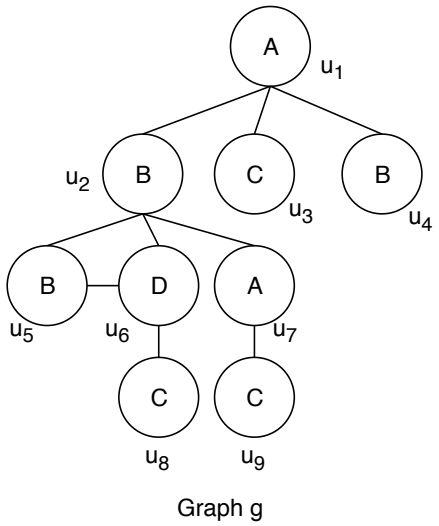


Figure 1.2: Graph representation : graph g

$$M = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

$$IA = \begin{bmatrix} 0 & 3 & 7 & 8 & 9 & 11 & 14 & 16 & 17 & 18 \end{bmatrix}$$

$$JA = \begin{bmatrix} 1 & 2 & 3 & 0 & 4 & 5 & 6 & 0 & 0 & \ldots \end{bmatrix}$$

$$LA = \begin{bmatrix} A & B & C & B & B & D & A & C & C \end{bmatrix}$$

1.4 CUDA Architecture

Figure 1.3 gives an overview of modern day GPU architecture. Kirk and mei W. Hwu [2010] has been referred for CUDA related concepts. CUDA enabled GPU is made up of number of highly threaded streaming microprocessors. In the figure, there are 2 streaming multiprocessors in each building block. Each of these streaming microprocessors is itself made up of streaming processors (6 SPs per SM in figure 2). These steaming processors share control logic and instruction cache.The global memory is made of DRAM and has very high bandwidth. Although the GPU global memory has higher latency as compared to the CPU RAM, highly parallel applications can compensate for the latency as it has higher bandwidth resulting in superior execution time.

1.5 Organization of the Thesis

Chapter 2 discusses the previous works in this area and related concepts. In Chapter 3, the parallel implementation is discussed with example. Chapter 4 discusses the comparative study of performances of the sequential and the parallel implementations. The datasets on with the study is done are also explained in this section. Chapter 5 concludes the work with analysis and proposes future work and optimizations that can be done to improve the speedup.

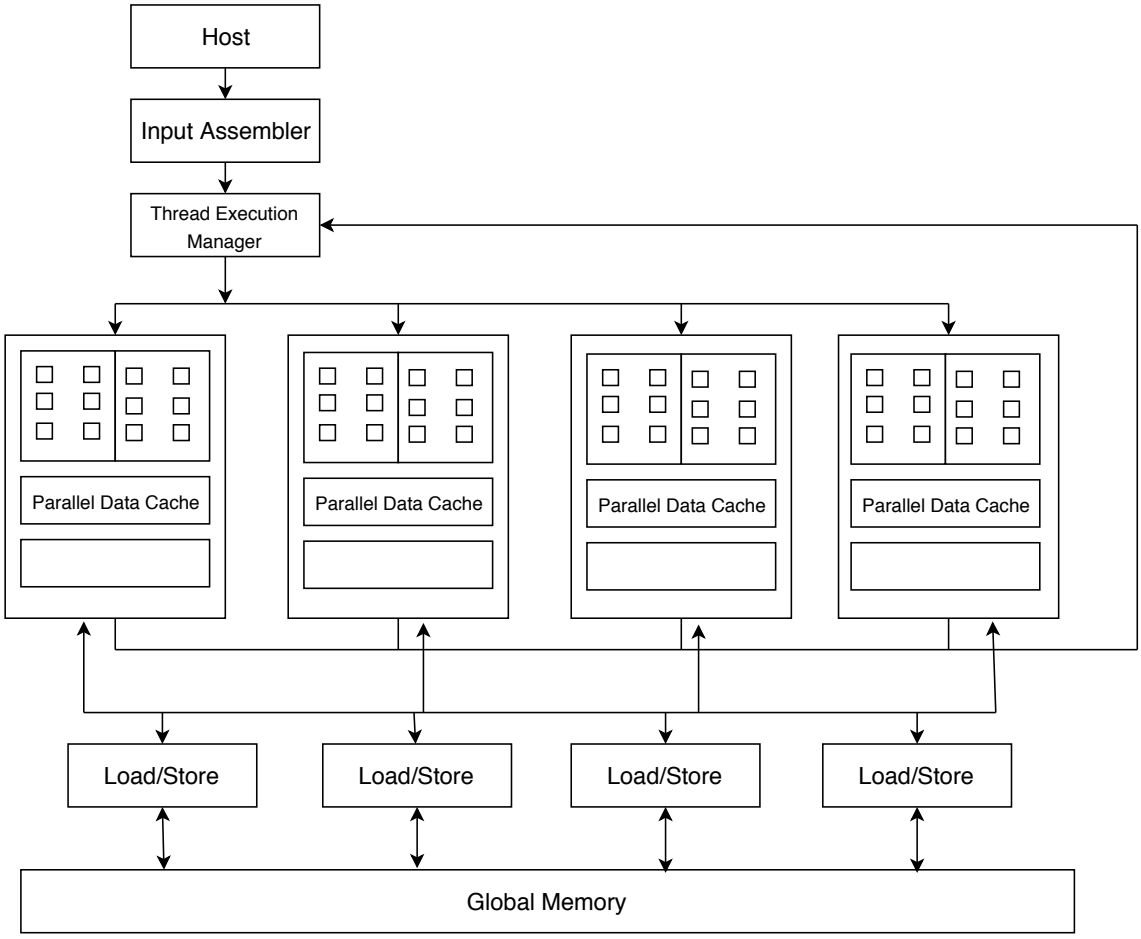


Figure 1.3: CUDA enabled GPU architecture

CHAPTER 2

LITERATURE SURVEY

This section discusses various state-of-the-art algorithms that have been developed. Ideas from these algorithms have been used in implementation of the parallel algorithm.

2.1 Generic Algorithm

The Generic Algorithm is template that is used most of the algorithms discussed in upcoming sections. Most of the popular algorithms are variations of the generic algorithm. It must be noted that the efficiency of algorithms depends on the heuristics used for pruning the candidate list and selecting the order of vertices that are to be matched. The Generic Algorithm uses backtracking for finding the solutions. If we find out that solution cannot be achieved from a call, we end that call and proceed to the next.

Algorithm 1 shows a generic subgraph isomorphism algorithm, GenericAlgo. Its inputs are a query graph q and a data graph g, and its output is a set of subgraph isomorphisms (or embeddings) of q in g. Here, to represent an embedding, we use a list M of pairs of a query vertex and a corresponding data vertex. The specific implementations of these procedures have been discussed in the respective subsections.

Algorithm 1 GenericAlgo

```
1: M ← ∅
2: for all u ∈ V(q) do
3:     C(u) ← FilterCandidates
4:     if c = ∅ then
5:         return
6:     end if
7: end for
8: SubgraphSearch
9: procedure SubgraphSearch
10:    if |M| = |V(q)| then
11:        report M
12:    else
13:        u ← NextQueryVertex(..)
14:        C_R ← RefineCandidates(...)
15:        for all V ∈ C_R such that v is not yet matched do
16:            if IsJoinable then
17:                UpdateState
18:            end if
19:        end for
20:    end if
21: end procedure
```

2.2 Ullmann Algorithm

Ulmann Algorithm was proposed in Ullmann [1976]. It exploits the template mentioned in Generic algorithm in the following way.

FilterCandidates: FilterCandidates returns a set that contains the vertices whose label set is a subset of label set of u. For example, in figure 1, it will return candidates of u_1 as $\{v_1, v_{1006}\}$

NextQueryVertex: Returns the next data vertex that is to be checked. Ullmann Algorithm takes the vertices the the same order in which they appear in the input. Hence the result of this Algorithm highly depends on the input order. This issue is discussed in the next subsection.

RefineCandidates: Prunes out all candidate vertices that have a smaller degree than u. In our running example, v_{1005} will be pruned out as a candidate for u_3 because it has degree less than that of u_3 and hence can never map to it.

IsJoinable: For all the vertices that have been matched, the IsJoinable checks whether there is a data edge corresponding to an edge in the query graph. In our running example, $\{u_1, u_2, u_3\}$ will be matched to $\{v_1, v_2, v_3\}$. The IsJoinable procedure will check if there are corresponding edges between these matches. Since the edge between v_2 and v_4 is absent, the IsJoinable procedure will return false.

2.3 VF2 algorithm

This algorithm was published by L. P. Cordella and Vento. [2004]. Following is the implementation of the algorithm in brief.

NextQueryVertex: VF2 Algorithm does not process data vertices in the order of input. Rather, from the start vertex, it returns the vertex which is a neighbour to the vertex (DFS like fashion). However, it does not use any particular method for selecting a neighbour. Hence, It does not exploit any property of the data/query graph.

RefineCandidates: VF2 uses the following pruning rules to prune out data vertex candidates:

1. If there is vertex v in the candidate list of a vertex u that is not connected to an already matched vertex, it is pruned out of the candidate list.
2. If there is vertex v in the candidate list of a vertex u, then the number of

unmatched vertices connected to v must be larger than those of u.

2.4 QuickSI Algorithm

This algorithm was proposed in H. Shang and Yu. [2008].

NextQueryVertex: The QuickSI Algorithm exploits properties of the data and query graphs in order to prune candidate list more quickly and get better matching sequence compared to the previously discussed ones. It preprocesses the data graphs for increased efficiency by maintaining 'score' for each vertex. Scores are calculated by assigning weights to each vertex in accordance to frequency of its label and degree of the vertex. Then the algorithm selects the vertices in the order in which they appear in the minimum spanning tree. This heuristic is very efficient as the low label frequency vertices have lower probability of getting a candidate and hence will get pruned earlier. In most of the cases, QuickSI gives a better result than Ullmann and VF2 algorithm when run sequentially.

2.5 Turbo_iso Algorithm

Algorithm 2 $Turbo_{ISO}(g, q)$

```
1: u_s ← ChooseStartQueryVertex(q,g)
2: q' ← ReWriteToNECTree(q, u_s) // q': NEC Tree
3: for all v_s in v|(v ∈ V(g)) and (L(u_s) ⊆ L(v)) do
4:     if ExploreCR(u'_s, v_s, CR) = FAIL then
5:         continue
6:     end if
7:     order = DetermineMatchingOrder(q', CR)
8:     UpdateState(M, F, us, vs)
9:     SubGraphSearch(g, q', q, order)
10:    RestoreState(M, F, u_s, v_s)
11: end for
```

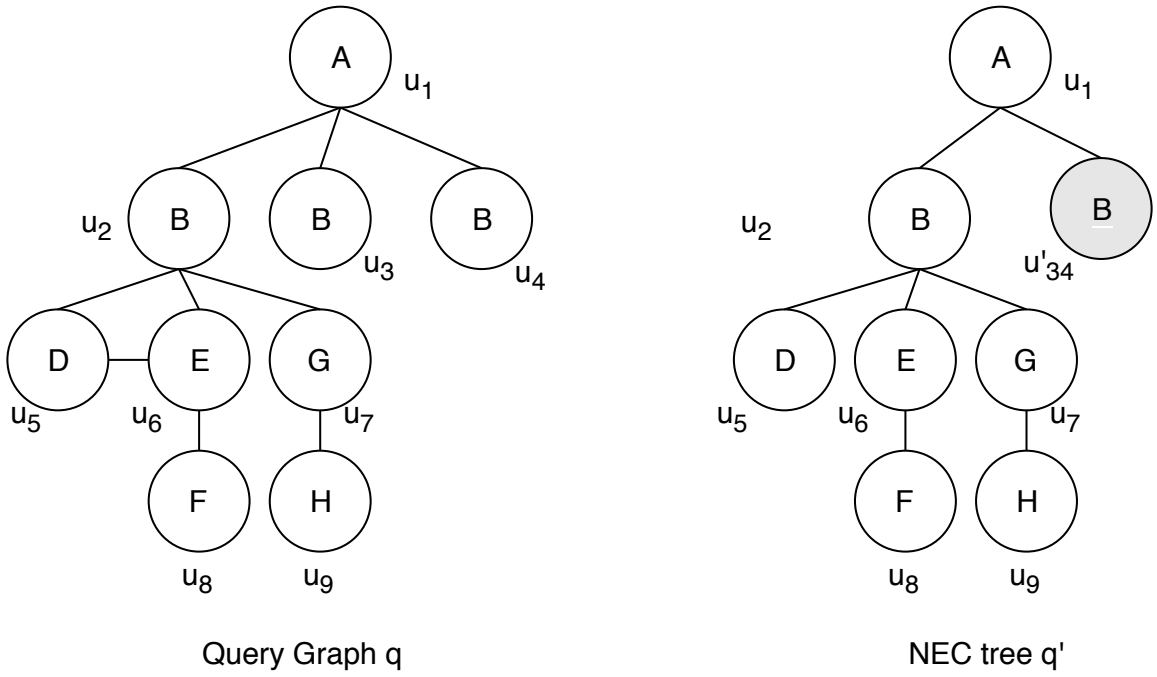


Figure 2.1: A query graph and its corresponding NEC Tree

This algorithm was proposed in Han *et al.* [2013]. $Turbo_{ISO}$ uses the concept of Neighbourhood Equivalence Class (NEC). Two vertices are said to be in an NEC if:

1. They have same labels.
2. The have exactly same subtrees (considering undirected graphs).

Figure 2.1 shows a query graph and it's equivalent NEC tree. In figure 2, u_2 and u_3 follow the above 2 rules and hence belong to the same NEC ($u'_{2,3}$). Matching time can be reduced using this concept as number of vertices that are to be matched will be reduced. Using NEC tree, we can reduce the number of call by huge margin as calls for vertices that are identically matching will be made only once. However, in order to generate all the matches, we will have to permute all the combinations for every NEC.

CHAPTER 3

Implementation

3.1 Data Pre-processing

In this section, we discuss various attempts made to reduce the search space and unnecessary recursion calls to save time and space. Following techniques have been used:

3.1.1 Candidate List Generation

Algorithm 3 getCandidates

Input: Data graph g, Query graph q, Data vertex v.
Output: Candidate set of all $u \in V(q)$

```
1: Candidate ← ∅
2:     ▷ For all v ∈ V(g), we launch a thread. Each thread performs the following operation
3: for all u ∈ V(q) do
4:     if degree(v) > degree(u) && L(u) = L(v) then
5:         index = atomicIncrement(numCandidates(u))
6:         Candidate(u, index) ← v
7:     end if
8: end for
9: return Candidate
```

This process helps to reduce the search space by identifying those vertices of data graph which can possibly be a match for any vertex of query graph. For a particular vertex u of a query graph, A vertex v of data graph is said to be a candidate iff:

- $|v| \geq |u|$
- $L(v) = L(u)$

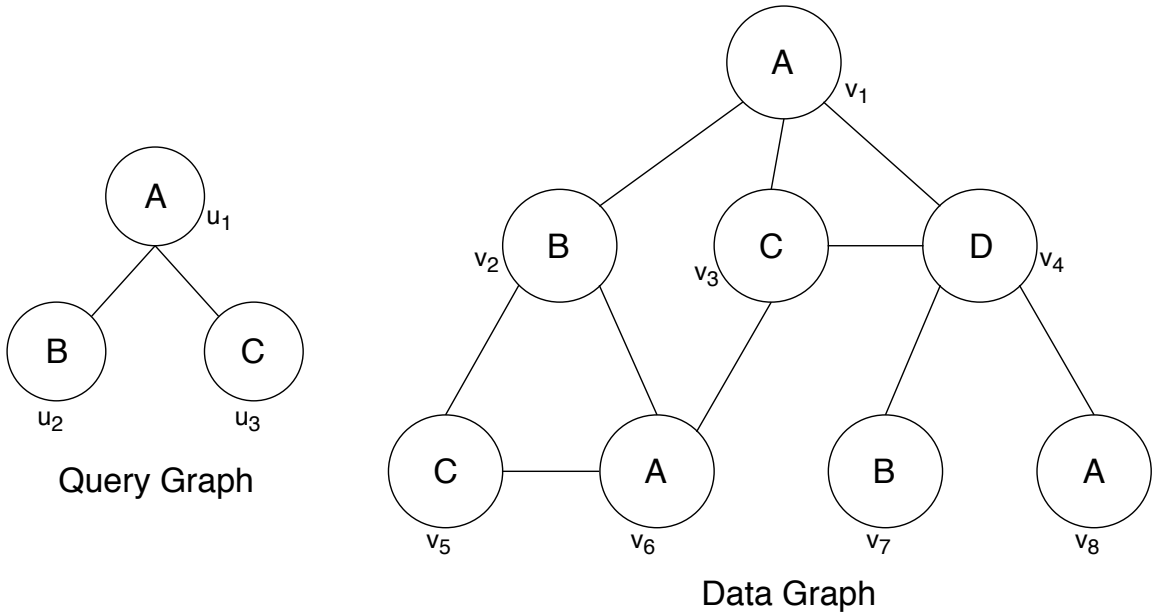


Figure 3.1: Data Graph g and Query Graph q

Consider the data graph and query graph as shown. As per the definition, the candidate list of $u_1 = \{v_1, v_6\}$, candidate list of $u_2 = \{v_2, v_7\}$ and $u_3 = \{v_3, v_5\}$.

In the parallel implementation of the method, we launch a thread for every vertex v of data graph. Each thread then finds all the query vertices of which v can be a candidate of. Since there is a data race between each thread to add a vertex $v \in V(d)$ to the candidate set of vertex $u \in V(q)$, atomicInc is used to prevent any data hazard due to the race. The numCandidate(u) returns the number of candidates of a vertex $u \in V(q)$ found so far.

3.2 Query vertex ordering

The order in which FindMatch processes the query vertex also determines the time of execution of algorithm. In order to have an efficient query vertex ordering, we suggest the following score function given to each query vertex:

$$score(u) = N(L(u), g)/|v|$$

where N(L(u), g) is the number of vertices in the data graph with the same label as that of u. The query vertices are then sorted in ascending order of their score. For e.g. for the graphs in figure 3.1, the order of query vertices will be u_1-u_2-u_3 with scores of 1.5, 2 and 2. The idea behind this scoring function is that higher the degree of a node, the more likely it is to get pruned while searching for a match. Inversely, the lesser the number of vertices in data graph with the same label as that of u, the more likely it will get pruned faster. However this may not always be the best choice.

3.3 Algorithm

In the parallel implementation of the Algorithm, first, we obtain all the candidate set of vertices of the query graph. Then we obtain the order in which we are to process the query vertices. These processes have been discussed in the previous section.

In the parallel implementation of findMatch, we select the first query vertex that is to be processed. For each candidate of the query vertex, we launch a thread that calls subgraphSearch. The subgraphSearch procedure recursively

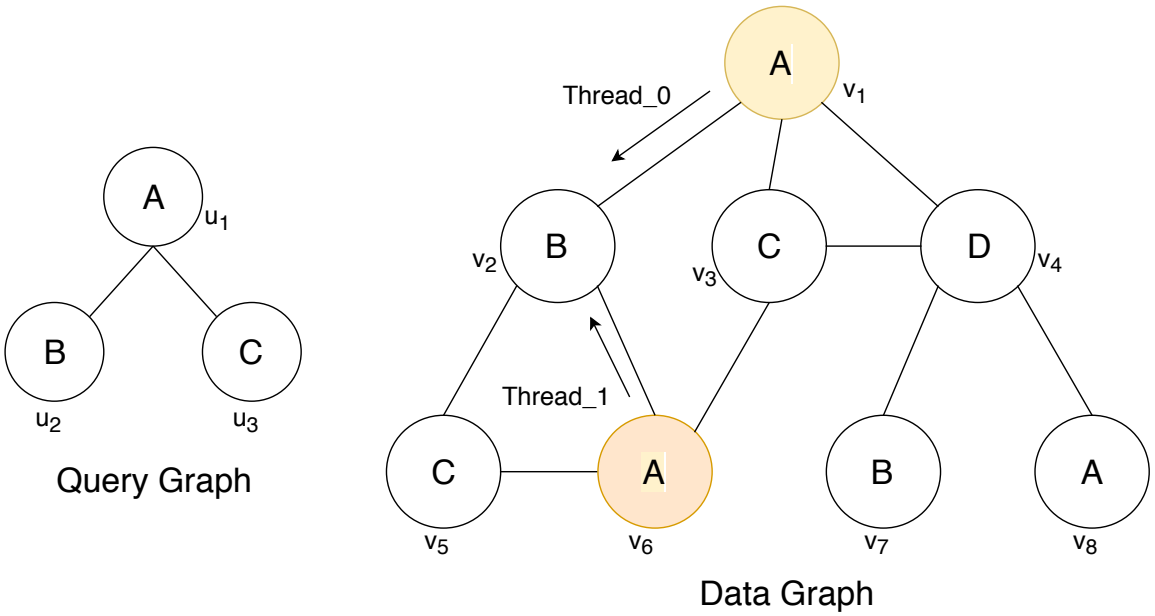


Figure 3.2: SubgraphSearch

checks for the candidates of the next query vertex that is to processed. If the candidate is joinable, i.e. the corresponding edges between the vertices in M and the corresponding vertices in q exist, then candidate is added to M.

For example, in the graph shown in figure 3.2, vertex u_1 is selected and two threads are launched which search for the match from vertices v_6 and vertex v_1 respectively. The threads then check for the candidates of vertices to follow. Say *thread*_0 is launched corresponding to the vertex v_1, it will check for the candidates of the vertex u_2. Since the corresponding edges between the vertices $\{u_1, u_2\}$ and $\{v_1, v_2\}$ exist, vertex v_2 is added to M and the state is updated. Similarly, since the corresponding edges between the vertices $\{u_1, u_2\}$ and $\{v_1, v_7\}$ do not exist, the vertex v_7 is discarded. Next, candidates of vertex u_3 are checked. In the graph, since the vertices $\{u_1, u_2, u_3\}$ and $\{v_1, v_2, v_3\}$ have the corresponding edges matched, the procedure SubgraphSearch reports it as a match. Since corresponding edges between $\{u_1, u_2, u_3\}$ and $\{v_1, v_2, v_5\}$ do not match, v_5 is discarded and the function returns.

Similarly, say $thread_1$ is launched corresponding to the vertex v_6. Similar procedure is performed i.e. vertex v_2 is added to M. In the next recursive call, both vertices v_3 and v_5 report a match. While the vertex v_7 is discarded because the corresponding edges do not match. Hence the the algorithm reports the match {v_1, v_2, v_3} (reported by thread_0), {v_6, v_2, v_5} and {v_6, v_2, v_3} (reported by thread_1).

.

Algorithm 4 FindMatch

Input: Data graph g, Query graph q, Candidate set C, Order of processing Query vertices
Output: all subgraph isomorphisms of q in g.

```
1: M ← ∅
2: C ← getCandidates
3: u ← NextQueryVertex
4: UpdateState(M, u, v...)
5:     ▷ For each u′ ∈ C(u), we launch a thread. each of which performs following operation
6: SubgraphSearch(u′, q, g, M, ...)
7: procedure SubgraphSearch(u, q, g, M, ...)[0]
8:     if |M| = |V(q)| then
9:         lock()
10:        report M
11:        unlock()
12:    else
13:        u′ ← NextQueryVertex
14:        for all v ∈ C(u′) do
15:            if ∼ IsVisited(v) then
16:                if IsJoinable(q, g, M, u, v) then
17:                    UpdateState(M, u, v...)
18:                    SubgraphSearch(u, q, g, M, ...)
19:                else
20:                    RestoreState(M, q, g, )
21:                    return
22:                end if
23:            end if
24:        end for
25:    end if
26: end procedure
```

3.4 Challenges

3.4.1 Recursive Implementation

Initially, the algorithm was implemented using recursion. However, since CUDA supports recursive implementation upto 24 levels, the algorithm did not work for query graph of size 8 vertices. To overcome this problem, the recursive procedure is implemented by maintaining stack. Following pseudocode shows stack implementation of SubgraphSearch

.

Algorithm 5 SubgraphSearch_Stack

```
1:  M ← ∅
2:  C ← getCandidates
3:  u ← NextQueryVertex
4:  push u
5:  while ~ stackEmpty do
6:      pop
7:      if |M| = |V(q)| then
8:          lock()
9:          report M
10:         unlock()
11:     else
12:         if ~ IsVisited(v) then
13:             if IsJoinable(q, g, M, u, v) then
14:                 UpdateState(M, u, v...)
15:                 push next candidate of current query vertex
16:                 push a candidate of next query vertex
17:             end if
18:         else
19:             push u
20:         end if
21:     end if
22: end while
```

3.4.2 Exponential Complexity

Because of the exponential complexity of the algorithm, a number of implementational challenges were faced. One of them being the use of dynamic parallelism. Since at each step number of calls increase exponentially, so does the number of threads. Hence dynamic parallelism is not used for implementing the algorithm.

CHAPTER 4

Experiments and Results

4.1 Datasets

The parallel implementation of Subgraph Isomorphism Algorithm has been compared to sequential implementation of VF2 algorithm on various real world datasets. Following is a brief overview of the datasets that have been used to perform experiments.

Table 4.1: Datasets

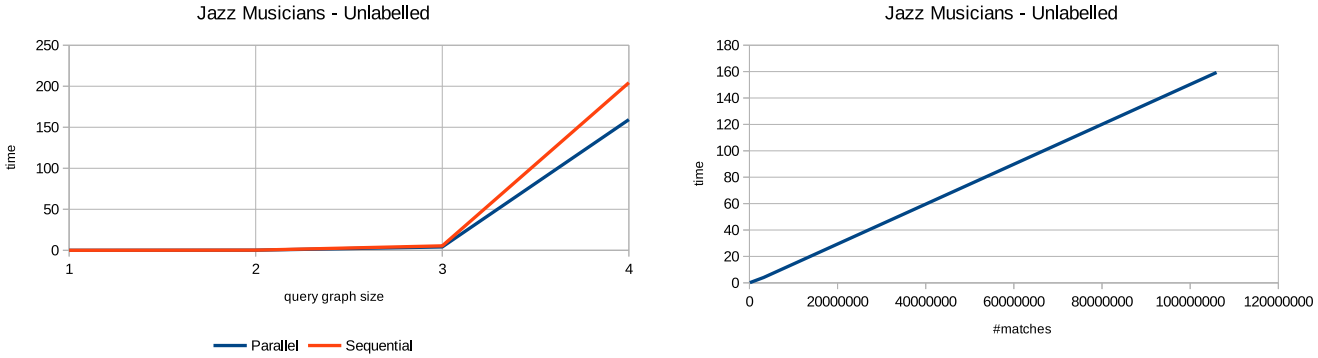
Dataset	**Vertices**	**Edges**
Jazz musicians	198	2742
Facebook (NIPS)	2888	2981
U. Rovira i Virgili	1133	5451
Hamsterster friendships	1858	12534
US power grid	4941	6594
Chicago transporation	1467	1298

For these datasets, random connected query graphs are generated to run the algorithm. To generate labelled graphs, labels are randomly allotted to the vertices of the data and query graphs.

4.2 Dataset: Jazz Musicians

The Jazz Musicians dataset (KONECT [*c*]) represents a network of musicians where each edge represents that the corresponding musicians have played together at

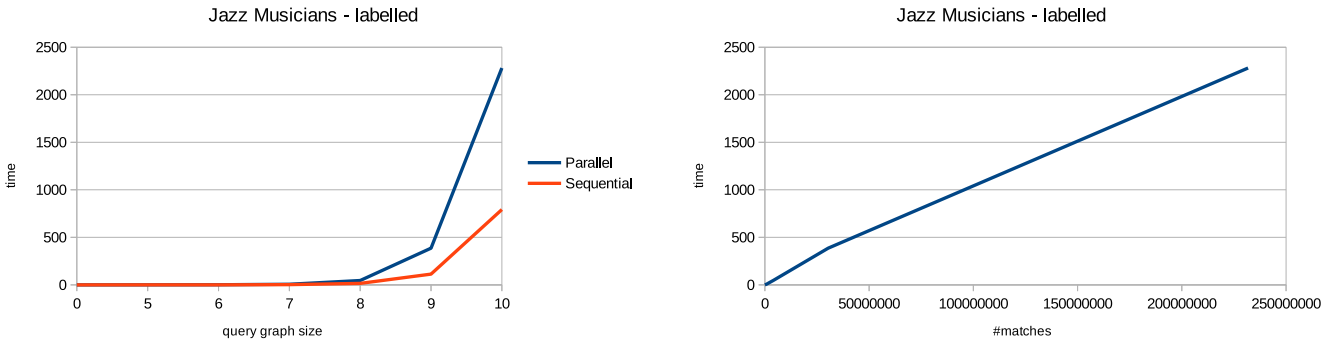
some point. This is a connected graph that consists of 198 vertices and 2742 edges. The average degree of this graph is 27.697 edges/vertex. For this dataset, we have compared the performance of the parallel implementation with the sequential implementation for both labelled and unlabelled graphs.



(a) average time vs query graph size (b) time vs #matches

Figure 4.1: Jazz musicians - unlabelled

As from the comparison graph in figure 4.1 (a), The parallel implementation runs faster as compared to the sequential version. However, both the implementations take exponential time to get the matches. In figure 4.1 (b), A plot for number of matches vs time elapsed is plotted. The number of matches obtained corresponds to various query graph sizes. We can observe that the time taken per match increases linearly (polynomially).



(a) average time vs query graph size (b) time vs #matches

Figure 4.2: Jazz musicians - labelled

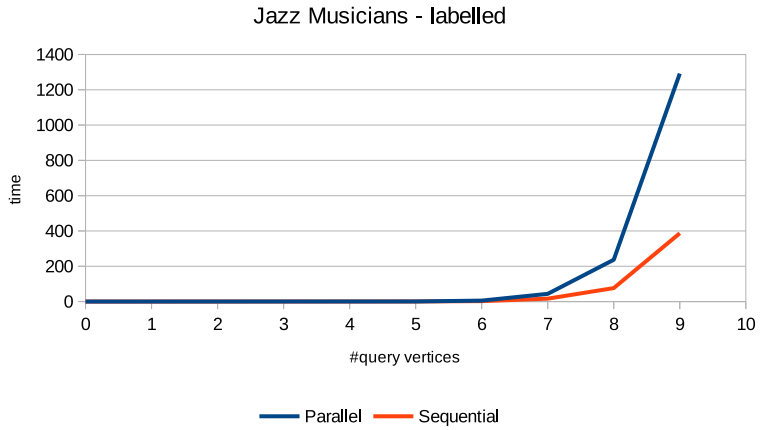


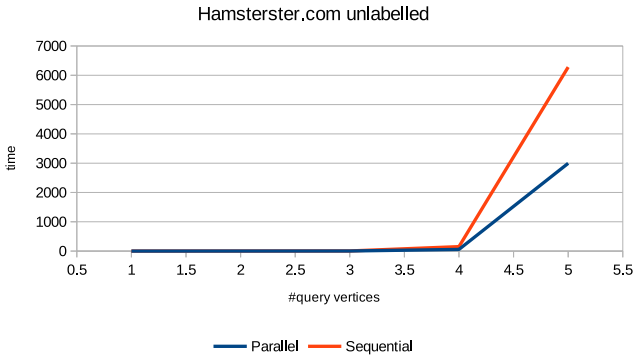
Figure 4.3: Jazz Musicians - labelled (dense)

For labelled graph, 5 random labels were assigned to the data and the query graphs. In this case, the performance of sequential implementation is better as compared to the parallel implementation. This is because in case of labelled graphs, the number of candidates of each node reduces. Because of which the number of threads and hence the amount of parallelism decreases.

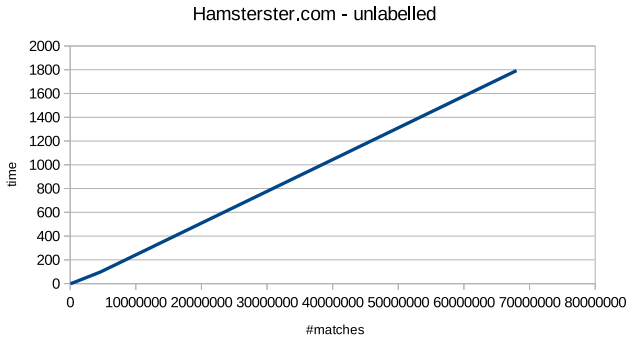
The algorithms were also compared on labelled data graph and dense labelled query graphs (both with 3 labels). But the result was similar as the sparse query graph.

4.3 Dataset: Hamsterster friendships

Source of this dataset is KONECT [*b*]. This graph represents network of friendship between users on the website hamsterster.com. The graph contains 1858 vertices and 12534 edges with a maximum degree of 272 edges and an average degree of 13.492 edges / vertex. Following plot shows the results of parallel and sequential implementation on the unlabelled graph.



(a) average time vs query graph size

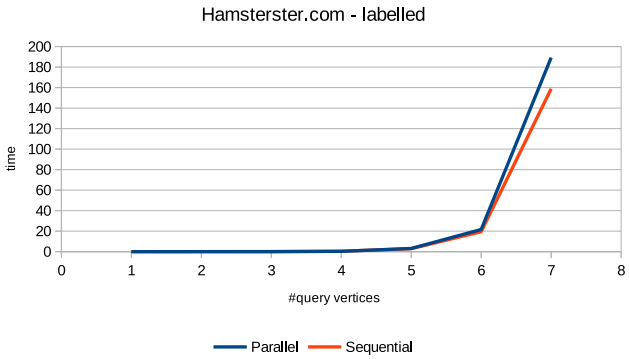


(b) time vs #matches

Figure 4.4: Hamsterster friendship - unlabelled

Since this graph is larger as compared to the jazz musicians graph, the number of candidates and hence the number of threads launched is more. Evidently, the performance of the parallel implementation is better than the sequential implementation. Also the average time elapsed per match is again linear.

As for the labelled graphs (5 labels), the performance of both the sequential and parallel versions are similar with the parallel implementation outperforming the sequential implementation for query sizes below 5 vertices.



(a) average time vs query graph size

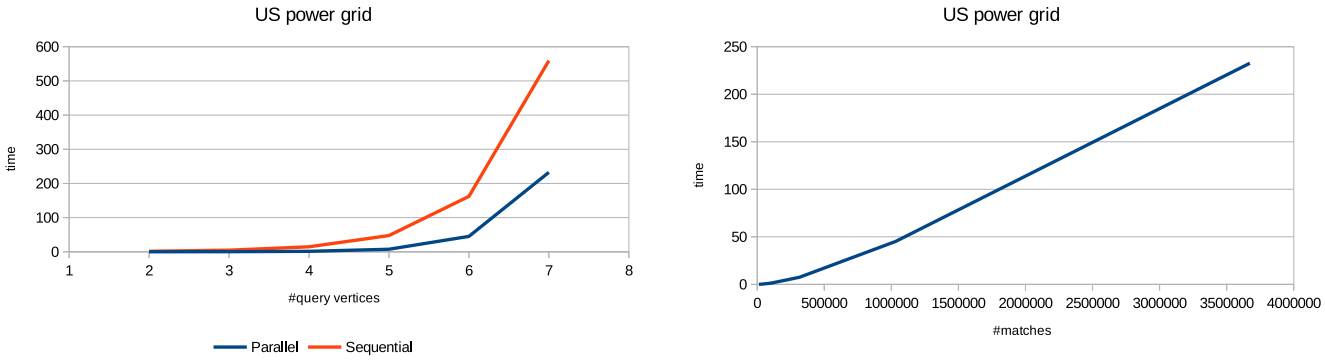
(b) time vs #matches

Figure 4.5: Hamsterster friendship - labelled

4.4 Dataset: US power grid

Source of this dataset is KONECT [*d*].This dataset contains the contains information on connectivity of the power grid of Western United States. Each node represents either a generator, a transformer or a substation. Each edge of the graph represents a wire connecting either of these. This graphs has 4941 vertices, 6594 edges and the maximum degree of a node is 19 edges. Since this a relatively sparse graph with an average degree of 2.6691 edges / vertex, the query graphs used on this data graph is relatively sparse in order to get sufficient number of matches.

Figure 4.6 (a) shows the comparison between performance of serial and parallel implementation of the algorithms over unlabelled data and query graphs.



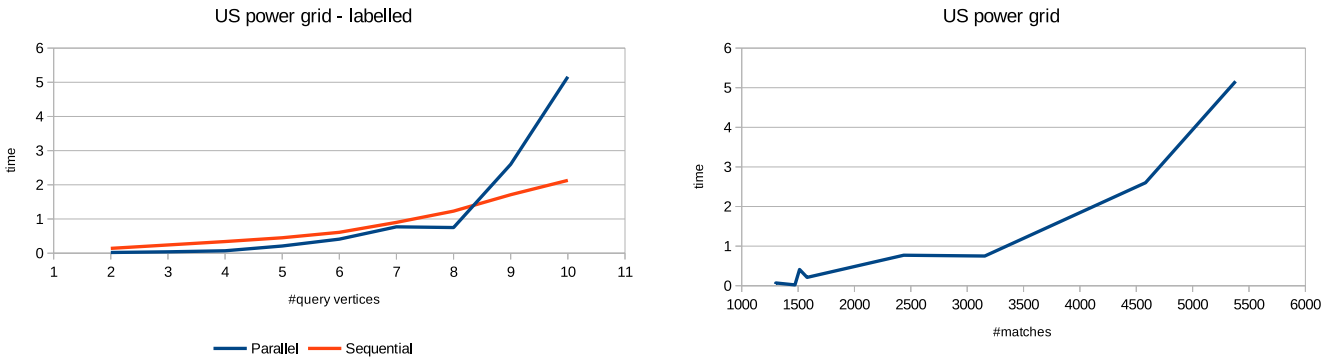
(a) average time vs query graph size (b) time vs #matches

Figure 4.6: US power grid - unlabelled

The performance of parallel implementation is better as compared to the sequential implementation for this dataset. The number of vertices in this graph is larger than the previous two graphs. The number of threads launched in the execution of parallel version for unlabelled graph is 3715.

Figure 4.7 (a) shows the comparison between performance of serial and parallel implementation of the algorithms over labelled data and query graphs. Since each node in the graph can either be a generator, a transformer or a substation, 3 labels

are used for the labelled graph which are randomly assigned to the nodes.



(a) average time vs query graph size (b) time vs #matches

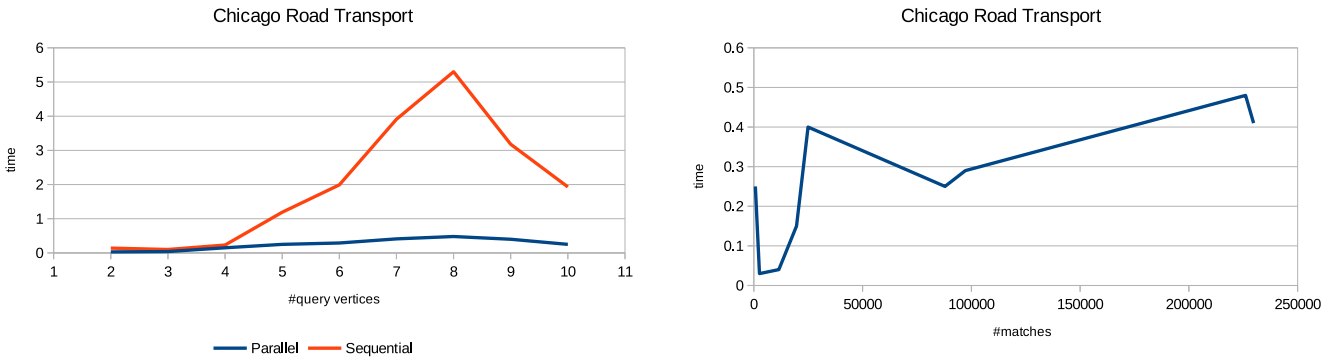
Figure 4.7: US power grid - labelled

In the labelled case, the parallel implementation performs better for query graph size ≤ 8. This is because, the number of candidates of each node decreases drastically with the increase in size of query vertex (given that data graph is very sparse). Another observation in this case is the irregular time / match curve. This is due to the fact that the number of matches is not monotonically increasing with the increase in query graph size, as was the case with all the previous graphs. The number of matches obtained for query size 2, 3, 4, 5 and 6 were 1472, 1299, 1303, 1580 and 1513 respectively.

4.5 Dataset: Chicago Transportation

Source of this dataset is KONECT [*a*]. This graph represents the transportation routes in Chicago city. The vertices in the graph represent a transportation terminal and an edge represents the connection between two terminals. This is a disconnected and a very sparse graph with 1467 vertices, 1298 edges with an average degree of 1.7696 edges / vertex and a max degree of 12 edges.

For this data graph, very sparse query graphs were used because the less number of matches with dense graphs (mostly 0). The comparison of performance is shown in figure 4.8



(a) average time vs query graph size (b) time vs #matches

Figure 4.8: Chicago transportation - labelled

Similar to the US power grid dataset, the time / match curve of this graph is very irregular because of the non monotonocity of the number of matches. For query size 2 to 10 in order, the number of matches are 2596, 11485, 19614, 87886, 97232, 229762, 226072, 24922 and 704. It is evident that the execution time of both the sequential and parallel algorithm depends on the number of matches. However, the parallel implementation outperforms the sequential implementation because of the comparatively large graph size.

CHAPTER 5

CONCLUSION AND FUTURE WORK

The parallel implementation of subgraph isomorphism outperforms the sequential implementation with the increase in size of data graph. As observed from the experiment, the parallel algorithm did not perform very well on small graphs like Hamsterster friendship and Jazz musicians. This is because the number of candidates and hence the number of threads launched were fewer. However, the parallel algorithm did perform well in larger, especially sparse datasets such as Chicago Transportation and the US power grid.

In future, further optimizations can be made to the parallel implementation such as load balancing between threads, to improve the performance even further. More complex heuristics can be developed in order to reduce the number of redundant matches that occur due to rotation and flipping of the same graph.

REFERENCES

H. Shang, X. L., Y. Zhang and **J. X. Yu.** (2008). Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 364–375.

Han, W.-S., **J. Lee**, and **J.-H. Lee** (2013). *turbo*$_{ISO}$: Towards ultrafast and robust subgraph isomorphism search in large graph databases. *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 337–348.

Jinsoo Lee, R. K., Wook-Shin Han and **J.-H. Lee** (2012). An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proceedings of the VLDB Endowment*, 133–144.

Kirk, D. B. and **W. mei W. Hwu**, *Programming on Massively Parallel Processors*. 2010.

KONECT (*a*). Chicago transportation. `http://konect.uni-koblenz.de/networks/tntp-ChicagoRegional`.

KONECT (*b*). Hamsterster friendship. `http://konect.uni-koblenz.de/networks/petster-friendships-hamster`.

KONECT (*c*). Jazz musicians. `http://konect.uni-koblenz.de/networks/arenas-jazz`.

KONECT (*d*). United states power grid. `http://konect.uni-koblenz.de/networks/opsahl-powergrid`.

L. P. Cordella, C. S., P. Foggia and **M. Vento.** (2004). A (sub)graph isomorphism algorithm for matching large graphs. *IEEE PAMI*, **26**, 1367–1372.

Shijie Zhang, S. L. and **J. Yang** (2009). Gaddi: Distance index based subgraph matching in biological networks. *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, 192–203.

Ullmann, J. R. (1976). An algorithm for subgraph isomorphism. *ACM*, **23**, 31–42.

Wikipedia (). Compressed sparse row (csr, crs or yale format). `https://en.wikipedia.org/wiki/Sparse_matrix`.