Parallel Algorithms for Scalable Graph Mining: Applications on Big Data and Machine Learning

Naw Safrin Sattar
*University of New Orleans, nsattar@uno.edu*

Follow this and additional works at: [https://scholarworks.uno.edu/td](https://scholarworks.uno.edu/td)

Part of the Artificial Intelligence and Robotics Commons, Computer and Systems Architecture Commons, Data Science Commons, Systems Architecture Commons, and the Theory and Algorithms Commons

**Recommended Citation**

This Dissertation is protected by copyright and/or related rights. It has been brought to you by ScholarWorks@UNO with permission from the rights-holder(s). You are free to use this Dissertation in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Dissertation has been accepted for inclusion in University of New Orleans Theses and Dissertations by an authorized administrator of ScholarWorks@UNO. For more information, please contact scholarworks@uno.edu.
Parallel Algorithms for Scalable Graph Mining: Applications on Big Data and Machine Learning

A Dissertation

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Engineering and Applied Science
Computer Science

by
Naw Safrin Sattar
M.S. University of New Orleans, May 2019
B.S. Bangladesh University of Engineering and Technology, April 2016

August, 2022
My dissertation work is wholeheartedly dedicated to my parents, Md Abdus Sattar and Syeda Suraiya Parveen who tried throughout their life to provide me with the best education possible.

To my beloved husband, Md Mohiuddin Sakib without whose encouragement and support it was not possible to aspire for higher studies at abroad. My super-smart kiddo, Sanaf Mahsoon, Mamma loves you a lot and you have been quite manageable and sensible at times when it was needed. I am thankful to my family for keeping faith in me and providing me with the best opportunity.

Last but not the least my mentor and PhD supervisor, Dr. Shaikh Arifuzzaman for his continuous help and support from the very beginning of my graduate study.
Acknowledgments

My heartiest gratitude to my PhD advisor Dr. Shaikh Arifuzzaman for his continuous support, enthusiastic participation and invaluable guidance throughout the work. I am grateful to my spouse Md Mohiuddin Sakib who continuously motivates me and provides me with mental support to carry on with my work. Without his suggestions and support, it would be near to impossible to take my degree to completion. I would also like to thank the members of my Dissertation Committee, Dr. Khaled Ibrahim, Dr. Dimitrios Charalampidis, Dr. Mahdi Abdelguerfi, and Dr. Tamjidul Hoque for their suggestions to improve the dissertation work. This work is partially supported by National Science Foundation (NSF) Grant 2132212, Louisiana Board of Regents RCS Grant LEQSF(2017-20)-RDA-25, UNO ORSP Award CON00000002410, and U.S. Department of Energy/Berkeley Lab Subcontract 7551418 (DoE Prime Contract No. DE-AC02-05CH11231). A part of the work has also been supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the US Dept. of Energy and the National Nuclear Security Administration. I am thankful to Dr. Aydin Buluc and Dr. Khaled Ibrahim from Lawrence Berkeley National Laboratory for their helpful guidance, suggestions and insights to improve my research starting from 2019 Summer internship and continuing the collaboration. I would like to thank my mentor Dr. Ying Wai Li from Los Alamos National Laboratory for her guidance and organized day to day activities to accomplish an interesting project during my 2021 Summer internship. Steve Spring and Chad Richards from Aspen Systems Inc. thank you for the opportunity to let me have the hands-on experience on the production of the supercomputers and sharing your expertise during my 2020 summer internship. My special thanks to all of my friends and well-wishers who have always been there for any kind of support any time.
# Table of Contents

List of Figures .................................................. ix
List of Tables ................................................... xiii
List of Algorithms ............................................... xv
Abstract ......................................................... xvi

Chapter 1 Introduction ........................................... 1
  1.1 Importance and Motivation .................................. 1
  1.1.1 Era of Exascale Computing and Big Data ............... 1
  1.1.2 Graph Mining and Graph Representation Learning ....... 2
  1.1.3 Machine Learning at Scale ............................... 3
  1.1.4 Big Graph Challenges .................................... 3
  1.2 Contribution ................................................ 4
    1.2.1 Parallel Algorithms for Community Detection on Large-scale Static and Dynamic Networks ............... 5
    1.2.2 Scalability of Neural Networks on GPUs ............... 7
    1.2.3 Real-world Big Graph Applications with Applied Machine Learning ........................................ 9
  1.3 Outline ...................................................... 12

Chapter 2 Scalable Distributed Louvain Algorithm for Community Detection in Large Graphs ........................................ 13
  2.1 Introduction ............................................... 14
  2.2 Related Works .............................................. 16
  2.3 Preliminaries .............................................. 18
    2.3.1 Notation ............................................... 18
    2.3.2 Louvain Algorithm for Community Detection .......... 19
      2.3.2.1 Modularity Optimization Phase .................... 20
      2.3.2.2 Community Aggregation Phase .................... 20
    2.3.3 Computational Model .................................... 21
    2.3.4 Graph Partitioner, METIS .............................. 21
  2.4 Methodology ............................................... 21
    2.4.1 Shared Memory Parallel Louvain Algorithm .......... 21
    2.4.2 Distributed Memory Parallel Louvain Algorithm .... 22
      2.4.2.1 Graph Partitioning ................................ 22
      2.4.2.2 Community Detection .............................. 22
    2.4.3 Hybrid Parallel Louvain Algorithm .................... 33
    2.4.4 Distributed Parallel Louvain Algorithm with Load-balancing ( DPLAL) ........................................ 33
  2.5 Experimental Setup ........................................ 35
    2.5.1 Execution Environment ................................ 35
    2.5.2 Description of Datasets ............................... 35
  2.6 Result ...................................................... 35
# TABLE OF CONTENTS

2.6.1 Speedup factors of shared and distributed memory algorithms .......................................................... 36
2.6.2 Speedup factors of our improved parallel algorithm DPLAL .............................................................. 37
   2.6.2.1 Runtime analysis: a breakdown of execution times ................................................................. 37
   2.6.2.2 Network size versus execution time. ...................................................................................... 38
   2.6.2.3 METIS partitioning approaches .......................................................................................... 39
2.6.3 Performance Analysis ......................................................................................................................... 39
   2.6.3.1 Comparison with Vite [1, 2] .............................................................................................. 41
   2.6.3.2 Comparison with Distributed Louvain by Charith et al. [3] ............................................ 42
2.6.4 Code Profiling: Performance Gain of DPLAL over Our Load imbalanced Naive Distributed Louvain Algorithm ............................................................................................................. 42
2.6.5 Discussion ....................................................................................................................................... 45

2.7 Conclusion ............................................................................................................................................ 46

Chapter 3 Understanding Evolving Community Structures in Temporal Graphs using Dynamic Community Discovery ............................................................................................................................. 47
3.1 Introduction ............................................................................................................................................ 47
3.2 Preliminaries ......................................................................................................................................... 49
   3.2.1 Notation ....................................................................................................................................... 49
   3.2.2 Temporal Community Measures .............................................................................................. 49
   3.2.3 Community Detection with Permanence .............................................................................. 49
   3.2.4 Computational Model ............................................................................................................... 51
3.3 Related Work ....................................................................................................................................... 51
3.4 Methodology ......................................................................................................................................... 52
   3.4.1 Community Evolution for Ground Truth Communities ............................................................... 52
      3.4.1.1 Pre-Processing Input Graph .............................................................................................. 52
      3.4.1.2 Mapping Ground Truth Communities per Snapshot ....................................................... 53
      3.4.1.3 Evaluating Community Structures using Graph Metrics per Snapshot .......................... 53
   3.4.2 Comparing Output Communities and Performance for Different DCD Algorithms ............... 53
      3.4.2.1 No Smoothing Louvain (NoSL) ...................................................................................... 53
      3.4.2.2 Smoothed Louvain (SmoL) ............................................................................................. 53
      3.4.2.3 DynaMo ........................................................................................................................... 54
      3.4.2.4 Estrangement Confinement(EsCon) .................................................................................. 54
      3.4.2.5 Transversal-Network(TraN) ............................................................................................ 54
      3.4.2.6 DyPerm ............................................................................................................................ 54
   3.4.3 Overview of the Shared-Memory Algorithm: ParDyPer ............................................................... 54
3.5 Experimental Setup ............................................................................................................................... 55
   3.5.1 Environment ............................................................................................................................... 55
   3.5.2 Dataset ........................................................................................................................................ 57
3.6 Result .................................................................................................................................................. 57
   3.6.1 Understanding the Evolution of Ground Truth Communities based on different Graph Metrics ........................................................................................................................................... 57
      3.6.1.1 Intra community edge ....................................................................................................... 60
      3.6.1.2 Internal Density ............................................................................................................... 60
      3.6.1.3 Average Degree .............................................................................................................. 61
      3.6.1.4 FOMD ............................................................................................................................. 62
      3.6.1.5 Inter Community Edge .................................................................................................... 62
      3.6.1.6 Expansion ......................................................................................................................... 63
      3.6.1.7 Cut Ratio .......................................................................................................................... 64
      3.6.1.8 Normalized Cut ............................................................................................................... 64
      3.6.1.9 Conductance ..................................................................................................................... 67
      3.6.1.10 Clustering Coefficient .................................................................................................... 67
      3.6.1.11 Separability ...................................................................................................................... 67
      3.6.1.12 Permanence .................................................................................................................... 67
   3.6.2 Comparison of Different Dynamic Community Detection Methods ............................................. 68
### Chapter 3 Parallel Dynamic Community Detection

- 3.6.3 Parallel Dynamic Community Detection: ParDyPer ........................................... 69
- 3.7 Conclusion ............................................................................................................. 70

### Chapter 4 DyGMPCD: A Distributed Parallel Community Detection Algorithm for Large-scale Dynamic Graphs

- 4.1 Introduction ........................................................................................................... 74
- 4.2 Related Works ...................................................................................................... 75
- 4.3 Preliminaries ......................................................................................................... 77
  - 4.3.1 Notation .......................................................................................................... 77
  - 4.3.2 Problem Description ....................................................................................... 77
  - 4.3.3 Community Detection with Permanence ....................................................... 77
  - 4.3.4 Computational Model ..................................................................................... 78
- 4.4 Methodology ......................................................................................................... 79
  - 4.4.1 Overview of our Distributed-Memory Algorithm .......................................... 79
  - 4.4.1.1 Graph Partitioning with METIS ................................................................. 80
  - 4.4.1.2 Community Detection ................................................................................. 80
  - 4.4.2 Implementation ............................................................................................... 83
    - 4.4.2.1 Challenges ................................................................................................ 83
    - 4.4.2.2 Selection of Efficient Data Structure ..................................................... 84
  - 4.4.3 Complexity Analysis ....................................................................................... 85
  - 4.4.4 Heuristics for Performance Optimization ...................................................... 85
- 4.5 Experimental Setup .............................................................................................. 86
  - 4.5.1 Environment .................................................................................................... 86
  - 4.5.2 Dataset ........................................................................................................... 86
- 4.6 Experimental Evaluation and Performance ......................................................... 87
  - 4.6.1 Scalability ........................................................................................................ 88
  - 4.6.2 Performance Gain using Different Heuristics ................................................. 89
    - 4.6.2.1 Heu-1: Pseudo Edge Detection with active/inactive labelled edges ........ 89
    - 4.6.2.2 Heu-2: Compressing Graph Size by Pruning Small Degree Nodes .......... 90
    - 4.6.2.3 Heu-3: Computational Efficiency by Excluding Smaller Communities .... 93
  - 4.6.3 Qualitative Analysis of the Community Structure .......................................... 94
  - 4.6.4 Comparison with the State-of-the-art Algorithms .......................................... 96
    - 4.6.4.1 STINGER shared-memory re-agglomeration algorithm ......................... 95
    - 4.6.4.2 Parallel Weighted Community Clustering (PWCC) Maximizing Spark-based Algorithm ................................................................. 96
  - 4.7 Conclusion .......................................................................................................... 96

### Chapter 5 Scalable Neural Networks

- 5.1 Community Detection using Semi-supervised Learning with Graph Convolutional Network on GPUs ........................................................................................................... 98
  - 5.1.1 Introduction ...................................................................................................... 98
  - 5.1.2 Related Works ................................................................................................. 100
  - 5.1.3 Preliminaries ................................................................................................... 101
  - 5.1.4 Methodology .................................................................................................. 102
    - 5.1.4.1 Computational Tools/Libraries ................................................................. 102
    - 5.1.4.2 Model Classifier ....................................................................................... 103
  - 5.1.5 Experimental Evaluation ................................................................................. 109
    - 5.1.5.1 Dataset ..................................................................................................... 109
    - 5.1.5.2 Experimental setting ............................................................................... 109
    - 5.1.5.3 Model Optimization ............................................................................... 109
    - 5.1.5.4 Comparison with the Related Methods ................................................... 117
    - 5.1.5.5 Model Performance Evaluation ............................................................... 117
  - 5.1.6 Conclusion ....................................................................................................... 118
- 5.2 Data Parallel Large Sparse Deep Neural Network on GPU .................................. 119
<table>
<thead>
<tr>
<th>Chapter 6 Big Graph Machine Learning Applications</th>
<th>133</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Detecting Web Spam in Webgraphs with Predictive Model Analysis</td>
<td>133</td>
</tr>
<tr>
<td>6.1.1 Introduction</td>
<td>134</td>
</tr>
<tr>
<td>6.1.2 Related Works</td>
<td>135</td>
</tr>
<tr>
<td>6.1.3 Preliminaries</td>
<td>136</td>
</tr>
<tr>
<td>6.1.3.1 Machine Learning Models</td>
<td>136</td>
</tr>
<tr>
<td>6.1.4 Methodology</td>
<td>139</td>
</tr>
<tr>
<td>6.1.4.1 Dataset</td>
<td>139</td>
</tr>
<tr>
<td>6.1.4.2 Feature Generation for Test Data</td>
<td>140</td>
</tr>
<tr>
<td>6.1.4.3 Environment</td>
<td>142</td>
</tr>
<tr>
<td>6.1.4.4 Building Classifier</td>
<td>142</td>
</tr>
<tr>
<td>6.1.4.5 Validation</td>
<td>143</td>
</tr>
<tr>
<td>6.1.4.6 Feature Selection</td>
<td>143</td>
</tr>
<tr>
<td>6.1.4.7 Testing Classifier with uk-2014 dataset</td>
<td>144</td>
</tr>
<tr>
<td>6.1.5 Result</td>
<td>144</td>
</tr>
<tr>
<td>6.1.5.1 10-fold Cross Validation Accuracy</td>
<td>144</td>
</tr>
<tr>
<td>6.1.5.2 F-measure</td>
<td>145</td>
</tr>
<tr>
<td>6.1.5.3 Finding Best Model for each Feature Set</td>
<td>145</td>
</tr>
<tr>
<td>6.1.5.4 Feature Selection for Improvement in Naïve Bayes Model</td>
<td>147</td>
</tr>
<tr>
<td>6.1.5.5 Graph based vs. Text based Feature Sets</td>
<td>149</td>
</tr>
<tr>
<td>6.1.5.6 uk-2014 Dataset Test Result</td>
<td>149</td>
</tr>
<tr>
<td>6.1.5.7 Weka versus Scikit-learn</td>
<td>150</td>
</tr>
<tr>
<td>6.1.5.8 Comparison with existing work</td>
<td>150</td>
</tr>
<tr>
<td>6.1.6 Conclusion</td>
<td>151</td>
</tr>
<tr>
<td>6.2 COVID-19 Vaccination Awareness and Aftermath: Public Sentiment Analysis on Twitter Data and Vaccinated Population Prediction in the USA</td>
<td>152</td>
</tr>
<tr>
<td>6.2.1 Introduction</td>
<td>152</td>
</tr>
<tr>
<td>6.2.2 Related Works</td>
<td>154</td>
</tr>
<tr>
<td>6.2.3 Preliminaries</td>
<td>156</td>
</tr>
<tr>
<td>6.2.3.1 Sentiment Analysis</td>
<td>156</td>
</tr>
<tr>
<td>6.2.3.2 Time Series Forecasting</td>
<td>157</td>
</tr>
<tr>
<td>6.2.4 Methodology</td>
<td>160</td>
</tr>
<tr>
<td>6.2.4.1 Computational Tools/Libraries</td>
<td>160</td>
</tr>
<tr>
<td>6.2.4.2 Environment</td>
<td>161</td>
</tr>
<tr>
<td>6.2.4.3 Sentiment Analysis</td>
<td>161</td>
</tr>
<tr>
<td>6.2.4.4 Forecasting Model for COVID-19 Vaccination in the USA</td>
<td>165</td>
</tr>
<tr>
<td>6.2.5 Experimental Evaluation</td>
<td>168</td>
</tr>
<tr>
<td>6.2.5.1 Public Sentiment on Vaccination</td>
<td>168</td>
</tr>
<tr>
<td>6.2.5.2 Public Sentiment on Healthy Lifestyle after Vaccination</td>
<td>172</td>
</tr>
<tr>
<td>6.2.5.3 COVID-19 Vaccination Forecast for the USA</td>
<td>178</td>
</tr>
<tr>
<td>6.2.6 Conclusions</td>
<td>183</td>
</tr>
</tbody>
</table>
**TABLE OF CONTENTS**

**Chapter 7  Conclusion** ................................. 189  
  7.1  Summary of Contributions ............................ 189  
  7.2  Future Research Directions ........................... 190  

**Bibliography** .............................................. 191  

**Vita** .................................................. 209
# List of Figures

1.1 Updates statistics of social and communication networks for 1 second of activity. ....... 4

2.1 A Sample Graph to explain the algorithmic steps used in Section 2.4.2. ................. 25

2.2 Speedup factors of our parallel Louvain algorithms for different types of networks. Our hybrid algorithm strikes a balance between shared and distributed memory based algorithms. .... 36

2.3 Runtime analysis of RoadNet-PA graph with DPLAL algorithm for varied number of processors. We show a breakdown of execution times for different modules or functions in the algorithm. Time for gathering updated communities and total duration are plotted w.r.t the right y-axis. .................................................. 38

2.4 Increase in runtime of DPLAL algorithm with an increase in the sizes of the graphs keeping the number of processors fixed (800 processors). ......................... 39

2.5 Comparison of METIS partitioning approaches (edge-cut versus communication volume minimization) for several networks. The edge-cut approach achieves better runtime efficiency for the above real-world networks. .................................. 40

2.6 Speedup factors of DPLAL and VITE algorithms for small networks. Speedup curve of Vite completely follows a decreasing trend, whereas DPLAL shows good speedup. ............... 40

2.7 Speedup factors of DPLAL and VITE algorithms for large networks. Speedup curve of Vite completely follows a decreasing trend, whereas DPLAL shows good speedup. ............... 41

2.8 Speedup factors of DPLAL and VITE algorithms for larger networks. Larger networks scale to a larger number of processors. In case of Vite, speedup increases up to 256 processors maximum and decreases afterwards. Speedup curve of DPLAL follows increasing trend. .... 41

2.9 DPLAL gains significant performance gain over our Naive Distributed Louvain algorithm. Here the functions are denoted as follows. func-1: gathering neighbour info func-2: exchanging updated community func-3: exchanging duality resolved community func-4: gathering updated communities .................................................. 43

2.10 Percentage of Time spent in the MPI Functions in DPLAL ........................................ 43

2.11 Runtime of MPI Functions in our initial Naive Distributed Louvain Algorithm ............. 44

2.12 Runtime of MPI Functions in DPLAL ................................................................. 44

2.13 MPI Communication pattern in our initial Naive Distributed Louvain Algorithm .......... 45

2.14 MPI Communication pattern in DPLAL ................................................................. 45

3.1 Calculation of Permanence using an example graph .................................................. 50

3.2 Workflow for Community Evolution and Parallel Community Detection in Temporal Graphs for different networks ................................................................. 51

3.3 Edge Addition, Edge Deletion and Vertices for different networks in different time-slices ... 55

3.4 Community Size (a-d) and Intra Community Edge (e-f) for different networks in different time-slices. For Intra Community Edge, Syn-1 and Syn-3 networks show constant insignificant values (0/1) over all snapshots. ........................................... 56

3.5 Community mapping using jaccard coefficient for the ground truth communities of Primary School network ................................................................. 57
3.6 Internal Density (a-c) and Average Degree (d-f) for different networks in different time-slices. For Cumulative CoAuthorship network the values of Internal Density are considered negligible being very small (0.01 – 0.15). Syn-3 Network follows the similar pattern as Syn-1 Network for the values of Average Degree.

3.7 FOMD for different networks in different time-slices

3.8 Inter Community Edges for different networks in different time-slices

3.9 Conductance for different networks in different time-slices

3.10 Clustering Co-efficient for different networks in different time-slices

3.11 Expansion, Cut Ratio and Normalized Cut for different networks in different time-slices. Both Syn-1 and Syn-3 networks has almost constant and very small values for all of these metrics and considered negligible

3.12 Separability (a-b) and Permanence (c-f) for different networks in different time-slices. Intra Community Edge being 0 for most communities in Syn-1 and Syn-3 networks, the value of separability also becomes 0 for those communities.

3.13 Comparison of different DCD methods for a particular community in Primary School Network

3.14 Comparison of different DCD methods for a particular community in Cumulative Co-authorship Network

3.15 Comparison of different DCD methods for a particular community in College Message Network

3.16 Comparison of different DCD methods for a particular community in Facebook Forum Network

3.17 Speed-up for shared-memory parallel algorithm ParDyPer for different networks

4.1 Calculation of Permanence using an example graph

4.2 Time (%) spent in dynamic edge updates from the overall computation time for the sorted and unsorted list using AL and PMA. The time is normalized w.r.t PMA.

4.3 Network statistics for different real-world dynamic networks in different time-slices

4.4 Percentage change of edge addition and deletion for different real-world dynamic networks in different time-slices

4.5 Speedup for our distributed parallel algorithm for multiple networks from different domains. For networks having edges more than a million, shows scalability for up to 1024 processors (maximum no. of processors used for experiments). For networks with less than a million edges, shows scalability for 64 processors only and do not scale more after this point.

4.6 Scalability of our parallel algorithm for increasing network size (number of edges) observed for 1024 processors

4.7 Performance gain of DyGMPCD using heuristic, Heu-1 observed for 1024 processors. Heuristic, Heu-1 is based on active/inactive edge labels to decide on the presence or absence of edges at each time slices. Green color denotes the improved speedup gained by running the Heu-1 heuristic based algorithm, whereas the base speedup shown with the orange color is obtained by running our main algorithm without using any heuristics.

4.8 ‘sx-stackoverflow’ is a representative network with huge number of smaller degrees (degree: 0-3) in most of the snapshots. The percentage of degree distribution according to number of degrees is given by the stacked bars in the left figure (a). In the right figure (b), the average degree of the network per snapshot is shown by the blue bars.

4.9 Performance gain of our parallel algorithm using heuristic, Heu-2 and combining both Heu-1 and Heu-2 heuristics observed for 1024 processors. The left figure (a) shows the change in speedup. The right figure (b) shows the performance gain in percentage.

4.10 ‘wiki-talk’ is a representative network with huge number of singleton communities in most of the snapshots. The percentage of community distribution according to community size is given by the stacked bars in the left figure (a). In the right figure (b), the average number of communities of the network per snapshot is shown by the blue bars.

4.11 Performance gain of our parallel algorithm using heuristic Heu-3 and the combination of all Heu-1, Heu-2 and Heu-3 heuristics observed for 1024 processors. The left figure (a) shows the change in speedup. The right figure (b) shows the performance increase in percentage.
4.12 Comparison of the output community quality in between the sequential and parallel methods based on permanence. Applying heuristic, Heu-1 keeps the permanence value unchanged as well.

4.13 Comparison of Parallel Performance based on Throughput [Updates per second by threads (OpenMP)/processors(MPI)] for coPapersDBLP network. Legends present different batch sizes. Left Figure (a) shows the throughput for STINGER parallel re-agglomeration algorithm. Right Figure (b) shows the throughput for our distributed parallel algorithm DyGM-PCD.

5.1 Change of accuracy with increasing label rate for identity matrix based feature set. The optimal label rate is in between the range 0.001 to 0.025 for all domains of network, to prevent the model from over-fitting.

5.2 Accuracy for different networks varying learning rate hyper parameter. For learning rate of 0.01, best accuracy is achieved for all network domains.

5.3 Accuracy for different networks varying dropout hyper parameter. For Citation, Communication and Internet Networks optimum dropout is 0.5. Except Road Networks, other domains have 0.1 optimum dropout.

5.4 Time-Efficiency with Increased Dropout Relaxing Accuracy for Networks from All Domains.

5.5 Accuracy for different networks varying the number of epoch. The optimal number of epoch for smaller, medium and larger networks are 200 to 250, 500 to 650 and 700 to 800 respectively.

5.6 Speed-up of our model for best performance on GPUs over CPUs. 4x speed-up is observed using 2 GPUs over CPUs.

5.7 Four layer (L = 4) deep neural network architecture. The input features $y^{(0)}$ are passed through a series of network layers $W^{(l=0,1,2,3)}$ with bias terms $b^{(l=0,1,2,3)}$, that produce scores for categories $y^{(L=4)}$.

5.8 Change of Rate with increasing computing power compared to the MATLAB baseline for different size of DNNs.

5.9 Speedup of DNNs in Python TensorFlow CPUs and GPUs over the baseline MATLAB Implementation [L denotes number of Layers].

5.10 Speedup of DNNs in GPUs over the CPUs using Python TensorFlow [N-Neuron, L-Layer].

5.11 Change in Runtime for Different Cross Device Communication Patterns for Distributed Mirrored Strategy [N-Neuron, L-Layer].

5.12 Change in Runtime for Different Cross Device Communication Patterns for Distributed Multi Worker Mirrored Strategy [N-Neuron, L-Layer].

6.1 Overview of Feature Generation Steps

6.2 Determining K value for the Feature Sets

6.3 Changes in Naïve Bayes Accuracy with Feature Selection

6.4 Comparison between Graph based (Link and Transformed Link) Feature Sets and Text based (Content) Feature Set

6.5 Performance Gain in SVM Model for Content Feature Set

6.6 Workflow for sentiment analysis on Twitter data using TextBlob and VADER.

6.7 Twitter data collection pipeline.

6.8 Workflow for COVID-19 vaccination forecasting model using WEKA.

6.9 Sentiment Distribution for Different Vaccines

6.10 Top Countries with Tweets for Twitter Dataset 1 (Different Vaccines) and Twitter Dataset 2 (Healthy Lifestyle).

6.11 Polarity Score of Sentiment for Different Vaccines

6.12 Subtlety Score of Sentiment for Different Vaccines

6.13 Top Frequency Word Distribution for different Vaccines: Word Cloud generated using word frequencies

6.14 Public Sentiment for different Vaccines with Timeline using TextBlob

6.15 Public Sentiment for different Vaccines with Timeline using VADER

6.16 Top Frequency Word Distribution for different Vaccines
6.16 Top Frequency Word Distribution for different Vaccines (contd.) 177
6.17 Public Sentiment Distribution for Twitter Dataset 2 on Healthy Lifestyle 178
6.18 Public Sentiment for Twitter Dataset 2 on Healthy Lifestyle with Timeline using TextBlob 178
6.19 Public Sentiment for Twitter Dataset 2 on Healthy Lifestyle with Timeline using VADER 179
6.20 Top frequency word distribution for Twitter Dataset 2 on healthy lifestyle 181
6.21 Combined top frequency words on Twitter Dataset 2 on healthy lifestyle 183
6.22 Polarity Score of Public Sentiment for Twitter Dataset 2 on Health 183
6.23 Subjectivity Score of Public Sentiment for Twitter Dataset 2 on Health 184
6.24 Top Frequency Word Distribution for Twitter Dataset 2 on Health: Word Cloud generated using word frequencies 185
6.25 COVID-19 vaccination prediction to forecast fully vaccinated population in the USA for different machine learning models with forecast time of 2 months 186
6.26 COVID-19 vaccination prediction to forecast partially vaccinated population in the USA for different machine learning models with forecast time of 2 months 187
6.27 COVID-19 Vaccination Prediction for different Machine Learning Models with Forecast Time of 1 week 188
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Terminologies used in explaining the algorithm</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>User-defined operators used in the algorithm</td>
<td>19</td>
</tr>
<tr>
<td>2.3</td>
<td>Symbols used for calculating Modularity in Equation 2.1 and Equation 2.2</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>Example Illustration for Send List and Receive List Count given in Step 3 [Algorithm 2]</td>
<td>26</td>
</tr>
<tr>
<td>2.5</td>
<td>Example Illustration for Community Duality Problem given in Step 7</td>
<td>30</td>
</tr>
<tr>
<td>2.6</td>
<td>Datasets used in our experimental evaluation</td>
<td>36</td>
</tr>
<tr>
<td>3.1</td>
<td>Temporal Community Measures Depending on Internal Connectivity Used for Community Evolution</td>
<td>49</td>
</tr>
<tr>
<td>3.2</td>
<td>Temporal Community Measures Depending on External Connectivity Used for Community Evolution</td>
<td>50</td>
</tr>
<tr>
<td>3.3</td>
<td>Temporal Community Measures Depending on Both Internal and External Connectivity Used for Community Evolution</td>
<td>50</td>
</tr>
<tr>
<td>3.4</td>
<td>Symbols used for calculating Permanence in Equation 2.1</td>
<td>51</td>
</tr>
<tr>
<td>3.5</td>
<td>Real-world Networks used in the Experiment</td>
<td>58</td>
</tr>
<tr>
<td>3.6</td>
<td>Synthetic Networks used in the Experiment</td>
<td>60</td>
</tr>
<tr>
<td>3.7</td>
<td>Runtime for 6 Different Dynamic Community Detection Algorithms</td>
<td>68</td>
</tr>
<tr>
<td>4.1</td>
<td>Terminology used in the paper</td>
<td>77</td>
</tr>
<tr>
<td>4.2</td>
<td>Symbols used for calculating Permanence in Equation 4.1</td>
<td>78</td>
</tr>
<tr>
<td>4.3</td>
<td>Real-world Static Networks with synthetic dynamic updates used in the Experiment</td>
<td>87</td>
</tr>
<tr>
<td>4.4</td>
<td>Real-world Dynamic Networks used in the Experiment</td>
<td>88</td>
</tr>
<tr>
<td>4.5</td>
<td>Deviation in permanence value after applying the heuristics, Heu-2 and both (Heu-1+Heu-2) together</td>
<td>94</td>
</tr>
<tr>
<td>4.6</td>
<td>Deviation in permanence value after applying the heuristics, Heu-1, Heu-2 and Heu-3 on wiki-talk network</td>
<td>94</td>
</tr>
<tr>
<td>5.1</td>
<td>Node-based Feature Set Generated Using Gephi</td>
<td>103</td>
</tr>
<tr>
<td>5.2</td>
<td>Additional Node-based Features Generated Using GraphVis</td>
<td>104</td>
</tr>
<tr>
<td>5.3</td>
<td>Networks Used in Experimentation</td>
<td>105</td>
</tr>
<tr>
<td>5.4</td>
<td>Performance Evaluation for Different Networks Based on Identity Matrix (I.M.) and Graph (G.) Features</td>
<td>108</td>
</tr>
<tr>
<td>5.5</td>
<td>Label Rate of Different Networks Used in Experiments for Both Identity Matrix Feature Set and Graph Feature Set</td>
<td>110</td>
</tr>
<tr>
<td>5.6</td>
<td>Performance Evaluation for Different Networks with Random Train-Test Distribution [I.M.-Identity Matrix, G.-Node-based Graph features]</td>
<td>111</td>
</tr>
<tr>
<td>5.7</td>
<td>Model Performance Improvement Using Changed Number of Labels Per Class for Node-based Graph Features</td>
<td>111</td>
</tr>
<tr>
<td>5.8</td>
<td>Change of Model Performance with Varying Number of Convolutional Layers</td>
<td>112</td>
</tr>
<tr>
<td>5.9</td>
<td>Comparison of Our GCN-based Method and Other Related Existing Methods [4, 5] on arxiv and PPI datasets.</td>
<td>117</td>
</tr>
<tr>
<td>5.10</td>
<td>Memory Requirement for various DNN size</td>
<td>126</td>
</tr>
<tr>
<td>5.11</td>
<td>MNIST Input Resizing</td>
<td>128</td>
</tr>
</tbody>
</table>
5.12 Computational Result on Different Sparse DNNs using MATLAB on CPU (Serial Implementation) .......................................................... 131
5.13 Computational Result on Different Sparse DNNs using Python TensorFlow on both CPUs and GPUs, Speedup calculated w.r.t. Baseline MATLAB Serial Implementation [*=Out of GPU Local Memory] .................................................. 131
5.14 Performance Comparison of Different Strategies for Distributed Training in TensorFlow .......................................................... 132

6.1 Kernel Functions of SVM .............................................................. 137
6.2 Distance Functions for KNN .......................................................... 137
6.3 Feature Sets used in our Experimentation ...................................... 140
6.4 10-fold Cross Validation Accuracy for Scikit-learn and Weka ............. 144
6.5 F-measure for Scikit-learn and Weka ............................................ 144
6.6 Machine Learning Model and Tool Selection for each Feature Set ........... 146
6.7 Feature Selection for Performance Gain ......................................... 147
6.8 10-fold Cross Validation Accuracy and F-measure with Link Feature Set and Transformed Link Feature Set for uk-2014 Test Dataset using Scikit-learn and Weka ............................ 149
6.9 Performance evaluation metrics for time series forecasting .................. 158
6.10 Twitter Dataset 1: Vaccination tweets used in our experimental evaluation. 163
6.11 Twitter Dataset 2: Tweets on healthy lifestyle used in our experimental evaluation. 164
6.12 Features related to COVID-19 vaccination. ...................................... 166
6.13 Tweets on Johnson & Johnson Vaccine on April 13, 2021 .................. 177
6.14 Random tweets from Twitter Dataset 2 on social distancing ............... 180
6.15 Prediction accuracy and error for different machine learning models to forecast fully vaccinated population in the USA ............... 182
6.16 Prediction accuracy and error for different machine learning models to forecast partially vaccinated population (at least one dose vaccine) in the USA for July 4, 2021 ............. 182
6.17 Prediction accuracy and error for different machine learning models to forecast partially vaccinated population (at least one dose vaccine) in the USA for 2 month time-frame ........ 182
List of Algorithms

1. Our Naive Distributed Parallel Louvain using MPI ........................................ 23
2. Gather_Neighbour_Info() ............................................................................. 25
3. Compute_Community() .................................................................................. 27
4. Update_Send_Receive_List() .......................................................................... 28
5. Exchange_Updated_Community() ..................................................................... 29
6. RCD() and EDRC() ........................................................................................ 30
7. Find_Unique_Communities() ........................................................................... 31
8. Compute_Modularity() .................................................................................... 31
9. DPLAL-Distributed Parallel Louvain Algorithm using Load-balancing .......... 34
10. DyGMPCD: Our MPI-based Parallel Dynamic Community Detection Algorithm ................................................................. 79
11. Edge Addition ................................................................................................... 80
12. Update Neighbors ............................................................................................. 81
13. Edge Deletion .................................................................................................... 82
14. non_unit_degree_edge_delete() .................................................................... 83
15. GCN using Mini-batch Gradient Descent ....................................................... 105
16. Serial Inference Calculation in MATLAB ..................................................... 122
17. A Pseudocode for creating custom layer to build our own model in TensorFlow .................................................. 125
18. Graph-based Feature Generation from Webgraph ......................................... 141
Abstract

Parallel computing plays a crucial role in processing large-scale graph data. Complex network analysis is an exciting area of research for many applications in different scientific domains e.g., sociology, biology, online media, recommendation systems and many more. Graph mining is an area of interest with diverse problems from different domains of our daily life. Due to the advancement of data and computing technologies, graph data is growing at an enormous rate, for example, the number of links in social networks is growing every millisecond. Machine/Deep learning plays a significant role for technological accomplishments to work with big data in modern era. We work on a well-known graph problem, community detection (CD). We design parallel algorithms for Louvain method for static networks and show around 12-fold speedup. The implementations use both shared-memory and distributed memory parallel algorithms. We also show the change of communities in dynamic networks in different time phases computing several graph metrics based on their temporal definition. We detect temporal communities in dynamic networks representing social/brain/communication/citation networks in a more concrete way. We present both shared-memory and distributed-memory parallel algorithms for CD in dynamic graphs using permanence, a vertex-based metric. The parallel CD algorithm implemented using Message Passing Interface (MPI) for temporal graphs is the first MPI-based algorithm to the best of our knowledge. Our algorithm achieves 30× speedup for the largest network with billions of edges. We present a scalable method for CD based on Graph Convolutional Network (GCN) via semi-supervised node classification using PyTorch with CUDA on GPU environment (4× performance gain). Our model achieves up to 86.9% accuracy and 0.85 F1 Score on different real-world datasets from diverse domains. We provide a scalable solution to the Sparse Deep Neural Network (DNN) Challenge by designing data parallel Sparse DNN using TensorFlow on GPU (4.7× speedup). We include the applications of webspam detection from webgraphs (billions of edges), sentiment analysis on social network, Twitter (1.2 million tweets) to reveal insights about COVID-19 vaccination awareness among the public and timeseries forecasting of the vaccinated population in the USA to portray the importance of graph mining in our daily activities.

Keywords: Graph Mining; Parallel Algorithms; Big Data; Scalability; MPI; OpenMP; GPU; Community Detection; Louvain Method; Temporal Network; Dynamic Network; Load-balancing; Graph Convolutional Network; Semi-supervised Learning; Deep Neural Network; Webgraph; Social Network
Chapter 1

Introduction

1.1 Importance and Motivation

1.1.1 Era of Exascale Computing and Big Data

We have now entered the era of exascale computing in recent times. United States has three of the five fastest supercomputers in the world according to the TOP500 supercomputer list published in November 2021. We need exascale computers to solve the challenges facing our world and the most complex scientific research questions need more and more computer power to solve. Exascale supercomputers will allow scientists to create more realistic models and provide solutions with a minimal time. The task of parallel computing is inevitable to utilize the processing power of the supercomputers efficiently. With the exponential growth of processing and network speeds, having only parallel architecture is not sufficient. We also need to perform parallel computations by fully utilizing the processing power of the supercomputers. Humans create 2.5 quintillion bytes (one quintillion byte equals one billion gigabytes) of information, send 333.2 billion emails, make 3 billion minutes of calls on Skype, share 5 billion Snapchat videos and photos every day. By 2025, it is expected that there will be around 200 zettabytes (one zetta byte equals one trillion gigabytes) of data [6]. The emergence of big data forces us to perform on trillions of data points at a time. Parallel computing is seen in the multi-core processors [7] in our phones and tablets that make them run faster. In the complex form, it is the stunning 200,000+ cores within the supercomputers that are helping the researchers and scientists to solve multi dimensional problems in the fields of health science, environmental modelling, astrophysics, quantum computing, finance and business. With the exponential growth of data, parallel computing will keep pace to assist us in working with large-scale dataset. Traditional data models and serial computing is
not sufficient enough to handle this overgrowing amount of data. New advanced data models such as graph based models are introduced to efficiently operate parallel processing on the data. Time is money. By saving time, parallel computing makes the work cost-effective. Serial computing forces fast processors to do things inefficiently. Instead of increasing cost on high configuration machines to run serial programs, we need to think of efficient parallel computation to utilize the processing powers of the supercomputers very effectively. Different parallel programming models comes handy to achieve the efficiency.

1.1.2 Graph Mining and Graph Representation Learning

Many data mining and machine learning (ML) challenges can be formalized as graph problems. Graph mining is the combination of tools and strategies utilized to analyze the properties of real-world graphs, predict how the structure and properties of a given graph might influence the real-world application, and create models that can produce practical graphs by the coordination of the designs found in important real-world problems [8]. Graphs are an omnipresent structure, utilized broadly inside computer science and related areas. Social networks, atomic structures, biological protein-protein systems, recommender systems and many other numerous domains can be promptly modeled as graphs. The omnipresence of graph makes it an important part of countless real-world systems to model by efficiently storing and accessing the relational knowledge of the system [9]. Graphs play an important role in modern machine learning. A neural network can be represented as a graph that makes predictions about other graphs. The vertices of the neural network are places where computation happens and the edges are the paths by which signal flows through the mathematical operations. Graphs have an arbitrary structure, with no specific beginning or end. For large social networks with billions of vertices, we cannot efficiently store such large network in a tensor. Neural networks work well on vectors and tensors for data types like images (fixed size and spatiality), text, time series (one direction). As a consequence, we need to think of representing or encoding graph structure in such a way that it can be easily exploited by machine learning models. In recent times, research has been going on the approaches that use techniques based on deep learning and nonlinear dimensionality reduction to automatically learn to encode graph structure into low-dimensional embeddings. The advancements in the area of representation learning on graphs primarily include matrix factorization-based methods, random-walk based algorithms, and graph convolutional networks [10]. Although some of the research involves generating representations that perform well on a particular set of classification or link prediction benchmarks, a consistent theoretical framework that
1.1.3 Machine Learning at Scale

In the arena of big data, the scalability of machine learning models is a prime concern for all. Coupling High Performance Computing (HPC) with Artificial Intelligence (AI) permits researchers to work on bigger/deeper, superior, and more precise machine learning models. Many real world applications require scaling ML models to multiple machines to deal with huge amount of data, achieve time and memory efficiency. Deep neural networks (DNNs) show groundbreaking advancement in the fields of computer vision, natural language processing, health science, recommender systems, and many other scientific domains. The compute requirements for model training grew by 300,000 times in the past few years [11]. To scale up the training process to parallel machines, data parallelism [12, 13, 14, 15] can be applied where mini-batch is partitioned among multiple workers. The model needs to be replicated on every processor. This method might not be the most space efficient but the design of such method is comparatively less complex. Another scaling technique is model parallelism where we need to split up the layers and run the model in a pipeline parallel mode. It requires communicating results after each group of layers. It saves memory but the design is more complex. Achieving balanced loads among multiple GPUs is very challenging. The complexity of different DNN layers varies, introducing significant efforts for programmers to partition model layers to GPUs in a balanced way. In order to achieve pipeline parallelism [16], understanding the mapping pattern of the communication is vital. Pipeline parallelism, itself, is a separate field of research. Another shortcoming of this method is to determine where the neural network is being split. The neural network requirements to process data also needs to be increased significantly. The weight staleness issue [17] is another concern. Since gradients are computed with stale weights, training instability and accuracy loss are persistent.

1.1.4 Big Graph Challenges

Big graphs analytic has drawn considerable attention to the researchers in both academia and industry. Facebook and Google have their own graph processing systems and use graph analytics in several use cases of their products. Large graphs can be in terabytes of size even when compressed and stored on disks. A single computer cannot process such large volume of data. When working with multiple processors, sometimes the data cannot be loaded completely to memory and requires distributed processing. The dynamic nature of several real-world graphs also pose a big challenge. In the traditional network mining, often the static topological properties of the graphs are used to solve different problems of graph theory. But time plays an important role to accurately represent the complex networks like social networks (Facebook, Twitter), brain networks, transportation networks and many more which change over time. 456,000 updates (tweets)
per minute are made daily over 199 million active vertices (users) in Twitter social network [6]. In email communication networks, 3.1 million emails are sent per second. Figure 1.1 shows the different update statistics per second in different networks [18].

![Figure 1.1: Updates statistics of social and communication networks for 1 second of activity.](image)

1.2 Contribution

In this dissertation we try to cover the broader domain of graph analytics focusing on the challenges related to big graphs and providing solution using different high performance computing techniques and platforms [19].

- Firstly, we work on developing scalable algorithms for a popular graph mining problem, “Community Detection” or “Clustering”. We design parallel algorithms for community detection in traditional static graphs in shared memory and distributed memory settings. Further, we extend our work for community detection in time-varying temporal or dynamic networks. We experiment and identify the best method to design parallel algorithm for temporal networks. The implementations use both shared-memory and distributed memory parallel algorithms.

- Secondly, we try to solve the same community detection problem using ‘graph convolutional network, as machine learning on graph is an interesting research problem drawing attention of a larger research community. Since deploying deep neural networks with limited resources is incredibly challenging, we also provide a scalable solution to the Sparse DNN Challenge—a challenge posed by MIT/IEEE/Amazon
GraphChallenge.org by designing data parallelism on GPUs.

- Finally, we apply machine learning on large-scale real-world graphs for important applications where we apply graph mining techniques to extract features from the networks.

1.2.1 Parallel Algorithms for Community Detection on Large-scale Static and Dynamic Networks

Community detection (or clustering) in large-scale graphs is an important problem in graph mining. Communities reveal interesting organizational and functional characteristics of a network. Community detection aims at detecting groups or hidden structures in a given graph [20]. Louvain algorithm is an efficient sequential algorithm for community detection. However, such sequential algorithms fail to scale for emerging large-scale data. Scalable parallel algorithms are necessary to process large graph datasets. We design parallel algorithms for Louvain method in shared memory and distributed memory settings. Developing distributed memory parallel algorithms is challenging because of inter-process communication and load balancing issues. We incorporate dynamic load balancing in our final algorithm DPLAL (Distributed Parallel Louvain Algorithm with Load-balancing). DPLAL overcomes the performance bottleneck of the previous algorithms and shows around 12-fold speedup scaling to a larger number of processors. We also compare the performance of our algorithm with some other prominent algorithms in the literature and get better or comparable performance. Majority of the works are shared memory based implementations. Often these implementations require highly configured systems for better scalability [21, 22, 23, 24, 25]. In our work we focus on clusters with moderate number of processors that is available to large number of researchers. Among very few works on distributed memory parallel algorithms, Vite, is the best performing distributed memory implementation of parallel Louvain algorithm [1, 2]. We have compared our distributed algorithm with Vite. Speedup largely depends upon the system architecture used for the experimentation because MPI performance varies with different system architecture [26]. As similar resource is not available to us during the experiment, we use our available resource, Louisiana Optical Network Infrastructure (LONI) QB2, which is a cluster containing 504 compute nodes (20 cores per node) with over 10,000 Intel Xeon processing cores of 2.8 GHz. On LONI, full usage of all the processors are not available to the users. So, we could use only 50 computing nodes at a time with 1,000 processors (maximum) for our experiments. We compare our algorithm with the baseline version of Vite, as we have not introduced any heuristics in the basic Louvain algorithm. For the same networks and system architecture, our algorithm achieves better speedup in most of the cases compared to Vite. We identify the challenges in developing distributed memory algorithm and pro-
vide an optimized solution DPLAL showing performance analysis of the algorithm on large-scale real-world networks from different domains.

We extend our study to temporal or dynamic networks. Temporal graphs are convenient mathematical abstraction for many practical complex systems. We present an analysis to see how communities evolve over times based on several graph metrics according to their temporal definitions. We also compare 6 different dynamic community detection (DCD) algorithms in terms of the quality of the output community and runtime. We find that the permanence-based method, DyPerm depending on a vertex-centric metric, works similarly efficient as DynaMo which is based on modularity. There exists several works on temporal community detection using different methods [27, 28] in literature. We find very few works similar to ours that do an extensive experimental analysis done by us in this study. Authors in [20] focus on progressively evolving graphs only in their study. They experiment on very small synthetic graphs with higher number of snapshots. On average those synthetic networks have only 100 nodes and 1200 steps (snapshots). They have compared the output community based on AMI, ARI. For AMI, ARI calculation what they consider as ground truth is not mentioned in the work. In our work we include both real-world as well as synthetic networks from different domains. In our analysis, we select networks from varied size range, starting with a few hundred nodes increasing up to millions of nodes and edges. We compare the community quality depending on 12 different graph metrics. [29] shows community quality evaluation at each time slice based on a single graph metric conductance. Other than this, no prior works analyzed the community structure quality over the snapshots. In our case we include 12 different graph metrics to measure how community evolution is taking place per snapshot. In [30] the authors have compared static community detection methods only for temporal social networks whereas we compare dynamic community detection methods for networks from multiple domains. Permanence allows for local optimization and such local computation helps in parallelizing the technique without incurring a significant communication overhead. We present a scalable algorithm, ParDyPer for community discovery in dynamic graphs and to the best of our knowledge, this is the first parallelization of permanence-based method for detecting communities in large dynamic graphs. Our shared-memory parallel algorithm achieves 4 − 18 fold speed-up for several real-world graphs from different domains as well as synthetic graphs.

We also present a novel parallel algorithmic framework, Dynamic Graph MPI-based Parallel Community Detection (DyGMPCD) for detecting communities in large-scale dynamic networks. To the best of our knowledge, there is no other MPI-based parallel methods in current literature for detecting communities in dynamic networks. Some recent works on dynamic graphs focus on providing a framework [31, 32, 33, 34, 35, 36] for storing and managing the networks with high performance and efficiency to support different graph
algorithms on GPUs and other various HPC platforms. Some of the parallel algorithms applied on dynamic networks include breadth first search (BFS) traversal [37], finding dense subgraphs [38], shortest path [39], k-cores [40] and others. However, very few works have been done on the parallel community detection algorithm for dynamic networks. Halappanavar et. al [41] have presented a technique to detect communities on dynamic graphs and future work includes implementing the method in their shared-memory based parallel framework Grappolo. The implementation and experimental analysis of their mentioned technique is not available yet. A shared-memory based multi-threaded community detection method for streaming graph is presented in [42]. The authors have implemented an incremental re-agglomeration algorithm using STINGER framework and Open Multi-Processing (OpenMP) that considers batch of changes to detect communities. The main limitation of the algorithm is that it cannot detect when a community is split into separate components. The only other parallel algorithm for detecting communities in dynamic networks is a Spark-based implementation that shows $2 \times$ parallel speedup with 12 cores [43]. Their algorithm identifies incremental vertices and defines community membership depending on maximizing Parallel Weighted Community Clustering (PWCC) metric of entire network. They have worked with mostly the synthetic networks with at most 2 million vertices and 10 million edges. The only real-world network they use have 200K vertices and 500K edges. For the synthetic networks, the degree distribution and the community size is very small which mostly does not represent the real-world dynamic networks like large-scale social networks those are highly skewed. We cannot compare our work with [43] as the source code is not publicly available. They have shown around $2 \times$ parallel speedup whereas for our largest network we get $30 \times$ speedup. We have compared our work with the existing OpenMP-based parallel algorithm [42] and our algorithm shows $30 \times$ performance gain over theirs. We also include 3 heuristics to our baseline algorithm. Applying these heuristics improve the performance of our algorithm significantly preserving the solution quality. We experiment on several real-world large-scale networks from different domains. DyGMPCD achieves up to $30 \times$ speedup for the largest network (0.2 billion edges).

### 1.2.2 Scalability of Neural Networks on GPUs

Graph Convolutional Network (GCN) has drawn considerable attention in recent times. GCN combines both graph algorithms and deep learning providing a vast area of research. GCN has several aspects in real-world. Many different problems of diverse domains can be solved efficiently using GCN. GCN is applicable to the following area such as citation network, social network, biological protein-protein interaction network, protein interface prediction [44], disease classification (PPIN Algorithm) [45], molecules in chemistry, text classification [46, 47], traffic congestion, image [48] classification [49, 50] and many more. Community
detection in graphs is a computationally challenging graph analytic problem. The presence of only a limited amount of labeled data (known communities) motivates us for using a learning approach to community discovery. However, detecting communities in large graphs using semi-supervised learning with GCN is an interesting research problem due to the scalability and accuracy issues. We present a scalable method for detecting communities based on GCN via semi-supervised node classification. We optimize the hyper-parameters for our semi-supervised model for detecting communities using PyTorch with CUDA on GPU environment. We demonstrate an experimental evaluation on different real-world networks from diverse domains. Our model achieves up to 86.9% accuracy and 0.85 F1-Score on these practical datasets. Our work is unique from others in the sense that we have used large-scale networks in our experiments and have not limited the number of communities as done in [5, 51]. We also compare our work with GraphSAGE [5] and our model shows better performance. Cluster-GCN [52] is a SGD-based (Stochastic Gradient Descent) algorithm to design the minibatches based on efficient graph clustering algorithms. GraphSAINT [53] is another graph sampling based inductive learning method that constructs minibatches by sampling the training graph, rather than the nodes or edges across GCN layers. Our work differs from these both [52, 53] in choosing the minibatches. In our work we do the sampling based on dense neighborhoods and keep track of the nodes in two consecutive layers. In our sample networks given in the experiments, the communities or labels vary within a large range, but these two methods experiment on networks with a limited number of labels. We apply Mini-batch Gradient Descent to solve the memory issues of GCN for larger and denser networks. We have done a comprehensive experimentations taking networks from diverse domains whereas only small-sized citation network is used in [54]. We also have not confined our work using the one-hot vector only. We have also used vertex based statics to generate graph feature set and made prediction on this feature set as well. In addition to that, we have made our model scalable to GPUs.

Further, we provide a scalable solution to the Sparse DNN Challenge—a challenge posed by MIT / IEEE / Amazon GraphChallenge.org—by designing model with data parallelism on GPUs. Since deploying deep neural networks with limited resources is very challenging, pruning the neural network efficiently is a solution to overcome this problem. Larger neural networks often perform better because larger number of layers/features allow more non-linear boundaries. However, such larger networks are constrained by large memory requirements. Sparse (pruned) neural networks deliver comparable performance with less amount of memory resources. Pruning [55] results in better generalisation results, improved speed of processing the results and a reduced size as well. The need for sparse DNNs has inspired the Sparse Deep Neural Network Graph Challenge by MIT/IEEE/Amazon GraphChallenge community. Sparse Deep Neural Network Graph Challenge performs neural network inference on a variety of sparse deep neural networks. We present a
solution to the Sparse DNN Graph Challenge using Python TensorFlow. We achieve up to 4.7-fold speedup using GPUs over the serial MATLAB implementation provided in GraphChallenge by implementing Sparse DNN in data parallel mode. We also compare different strategies of Distributed Training API of TensorFlow and demonstrate their performance empirically. A solution to the Sparse DNN Challenge has been provided by Davis et al. [56] using GraphBLAS [57]. The sequential performance of the GraphBLAS solution is $3 \times$ to $5 \times$ faster than the MATLAB reference implementation. Another Sparse DNN Challenge solution given in [58] shows a GPU implementation of the GraphBLAS standard. Their implementation shows a $1.94 \times$ speedup over the “SuiteSparse” CPU implementation of GraphBLAS. Our work is different from the works of [56, 58] as we focus on an efficient data parallel implementation using Python. Several deep learning frameworks, i.e., TensorFlow [59], PyTorch [60], are written in Python. Many scientific applications use these widely used deep learning frameworks. GraphBLAS focuses mainly on graph algorithms and is used by a specific community of graph researchers. Our work is generic in nature—it will help the end-users of different domains to apply these techniques with ease and achieve high performance capability.

1.2.3 Real-world Big Graph Applications with Applied Machine Learning

We provide solution to two important applications of large-scale real-world networks. Webgraphs can be exploited in detecting web spam. Webgraph is a graph having static HTML pages as nodes (vertices) and directed hyperlinks among the pages as edges [61]. In graph (network) mining, computing various structural properties of webgraphs is challenging due to the size of such graphs (billions of vertices and edges). Webgraph has great research potentials concerning web security. Webgraph is a potential source of detecting web spam based on graph based features. Emerging graph mining techniques can be used to detect spam in a scalable manner considering the large size of webgraph. Triangle count, clustering coefficient, triangular density, vertex jaccard similarity, vertex cosine similarity, and centrality measures are among potential features of either pages or hyperlinks to be used as features calculated from webgraphs rather than the contents of the pages. During our study, we face the difficulty to find labelled data of spam/non-spam. It shows the need for a machine learning classifier to predict spam based on the currently available labelled dataset. We generate graph-based feature set from webgraph for our test dataset. This can be applied to any webgraph for feature generation. Our model is tested on different dataset and achieve around 94% accuracy. We compare how the performance vary between graph based features and text based features. Very few works focus on webgraph’s graph properties to detect web spam. Many works [62, 63, 64] have been done using WEBSPAM-UK2007 for spam detection recently but different from ours. Their work did not emphasize on the graph-based features generated from the webgraph. Iqbal et. al [63] showed that Random Forest is the
best classifier for the given dataset. But they have not provided the detailed implementation (i.e. values for the parameters of the machine learning models used) for their work. [65, 66] also contributed to detect web spam using different machine learning models and datasets. Nevertheless none of those works used a different test dataset to show the test accuracy of the models. Besides, they used pre-computed feature sets for the labelled dataset and did not provide any details how to generate features for different data to test their models. They did not test their model on a different dataset and reported training accuracy only. We face difficulty to compare our work with the existing works as most of the works did not provide the model parameters for reproducibility. Again, the intermingling of feature sets do not match. We then compare the performance of our machine learning classifier with an existing work [64], although some information was missing. The authors developed SVM with two kernels. We have achieved a performance gain in simple SVM with rbf kernel. We have tested our model on a different most recently available dataset, uk-2014 on the same uk domain, and achieved at most 94% accuracy.

Later, we work on social network Twitter to reveal insights about COVID-19 vaccination awareness among the public. we work with a database of around 1.2 million tweets collected across five weeks of April-May, 2021 to draw conclusions about public attitudes towards the vaccination outlook when vaccination gets available to the mass people during the COVID-19 pandemic. We deploy natural language processing, and sentiment analysis techniques to reveal insights about COVID-19 vaccination awareness among public. There have been several works related to analyzing twitter dataset on different topics during COVID-19 pandemic [67, 68, 69, 70], but a few work focus on the twitter data related to COVID-19 vaccination [71, 72]. Authors in [67] have done text mining to identify addiction concerns during this COVID-19 pandemic. Their dataset has been concise to 3,301 tweets only. In [68] the authors have worked with Twitter data related to "Mask". [71] is based on sentiment analysis of COVID-19 vaccination tweets in Philippines. The authors have used Naïve Bayes model to classify English and Filipino language tweets (993 tweets) using the RapidMiner data science software with 81.77% accuracy. [72], analyzes if people are in the favor of receiving COVID-19 vaccine. Their result shows that average number of people have weakly positive sentiment in favor of having the COVID-19 vaccine shots. But the authors in their analysis have used a very limited number of tweets, only 900 tweets. They have not disclosed how they selected those tweets or what things they take into consideration while scraping those tweets. The timeline of scraping the tweets is also not mentioned in the study. Only [71] and [72], can be somehow related to our work. But [71] is related to tweets in Philippines only, whereas we collect tweets around the globe. For this reason, we have tweets around 1.2 millions, but they show analysis for 993 tweets only. This work also use Naïve Bayes model to predict the classification, whereas we classify the tweets with lexicon-based classifier and use the publicly available tools TextBlob and
VADER. For [71] manual annotation is done for the training data, i.e. manually providing the sentiment labels for the training data to predict the test data. We do not predict the sentiment labels, rather calculated labels using the well accepted sentiment analysis tools (TextBlob and VADER). So, we could not show any accuracy comparison with [71]. For another work given in [72], the twitter data collection criterion and timeline is missing, which is necessary if we want to compare our result with theirs. Their dataset consists of 900 tweets only and is also not publicly available, so we could not compare our work with theirs. We have done a thorough study on people’s sentiment about the COVID-19 vaccines and if they are maintaining a healthy way of life after getting vaccinated. We have chosen 7 different vaccines and collected tweets when vaccines are more accessible to general public. Our work is reproducible given our implementation details and publicly available code-base. Our result shows that people have positive attitude towards taking COVID-19 vaccines instead of some adverse effects of some of the vaccines. We also analyze people’s attitude towards the safety measures of COVID-19 after receiving the vaccines. Again, the positive sentiment is higher than that of negative in terms of maintaining safety measures against COVID-19 among the vaccinated population. Our another contribution of this study is related to time-series forecasting of vaccinated population in USA. Although there is a rich literature on different time-series forecasting methods for many different applications [73, 74, 75], no prior work is relevant to COVID-19 Vaccination Forecast. The study done by the Centers for Disease Control and Prevention (CDC) [76] predicts the cumulative death for COVID-19, 4 weeks ahead using the ensemble method in R. This prediction takes into consideration (e.g., COVID-19 data, demographic data, mobility data), methods, and estimates of the impacts of interventions (e.g., social distancing, use of face coverings) etc and not a time-series forecast. Another prediction of COVID-19 deaths and cases in the 15 countries of South and Central Europe has been done in [77] using Ensemble learning of the well known regression methods in WEKA. The dataset and detailed implementation is not well-described, so we cannot check how well their classifier could work on the vaccination dataset. We have not found any work that has shown forecast with vaccination data in USA. So, we could not show a comparison of our model in terms of accuracy with prior works. We have done a time-series forecasting on the US population to show what percentage of the population will be vaccinated at a near future time. We project that around 62.44% and 48% of the US population will get at least one dose of vaccine and be fully vaccinated respectively by the end of July 2021 according to our forecast model. Our prediction model gives a similar estimate of having partial vaccination of the adults to be 73.53% that the US Government is projecting to be 70% on the Independence Day July 4, 2021. This study helps to understand public reaction and be helpful for the policy makers to project the vaccination campaign as well as healthy and safety measures in the ongoing global health crisis.
1.3 Outline

This dissertation is made up of seven chapters and organized as follows. In Chapter 1 we discuss the importance and motivation of our work, describe our research directions and contributions. In the first part (Chapters 2, 3, 4), we discuss about our first research problem, design of parallel algorithms for community detection on large-scale static and dynamic networks. Particularly, in Chapter 2, we discuss the scalable Louvain algorithm for community detection on static networks. In Chapter 3, we discuss about the evolving nature of the temporal communities and the shared memory parallel implementation of community detection algorithm for temporal networks. In Chapter 4, we discuss the design of our distributed memory parallel community detection algorithm for temporal networks. We discuss about our second research problem, scalability of neural networks in Chapter 5. In section 5.1, we discuss about our scalable method for detecting communities based on GCN via semi-supervised node classification. Our scalable solution to the Sparse DNN is presented in Section 5.2 of the same chapter. Later in Chapter 6, we discuss about the real-world applications of big graphs, our third part of research contribution. In Section 6.1, we discuss an important application of web spam detection from webgraph. We also discuss about our work of sentiment analysis on social network Twitter and time-series forecasting of the COVID-19 vaccinated population in the USA in Section 6.2 of the same chapter. Finally, we conclude the dissertation in Chapter 7 by summarizing our contributions and highlighting some future prospects of the research work.
Chapter 2

Scalable Distributed Louvain Algorithm for Community Detection in Large Graphs

Community detection (or clustering) in large-scale graphs is an important problem in graph mining. Communities reveal interesting organizational and functional characteristics of a network. Louvain algorithm is an efficient sequential algorithm for community detection. However, such sequential algorithms fail to scale for emerging large-scale data. Scalable parallel algorithms are necessary to process large graph datasets. In this work, we show a comparative analysis of our different parallel implementations of Louvain algorithm. We design parallel algorithms for Louvain method in shared memory and distributed memory settings. Developing distributed memory parallel algorithms is challenging because of inter-process communication and load balancing issues. We incorporate dynamic load balancing in our final algorithm DPLAL (Distributed Parallel Louvain Algorithm with Load-balancing). DPLAL overcomes the performance bottleneck of the previous algorithms and shows around 12-fold speedup scaling to a larger number of processors. We also compare the performance of our algorithm with some other prominent algorithms in the literature and get better or comparable performance. We identify the challenges in developing distributed memory algorithm and provide an optimized solution DPLAL showing performance analysis of the algorithm on large-scale real-world networks from different domains.
2.1 Introduction

Parallel computing plays a crucial role in processing large-scale graph data [78, 79, 80, 3]. Complex network analysis is an exciting area of research for a large number of applications [81, 82]. The problem of community detection in complex networks arises in many scientific domains [83], e.g., sociology, biology, online media, recommendation systems, transportation and many more. Human brain can be represented as brain graphs where clustering of neural units into densely interconnected group is an useful application of community detection [84]. Such interconnection helps to determine the coordinated activity such as perception, action and adaptive behaviors. In recent times, community detection is very useful in preventing epidemic spreading [85] or false news propagation [86]. Fraud detection or detecting criminal activities in social networks is another application area for community detection [87, 88, 89]. Smart advertising and recommendation systems are a part of our daily life nowadays. These advertisements or recommendations are based on the like-mindedness of customers or users based on different communities [90, 91].

There are several measures and approaches to detect communities and evaluate the partitioned communities [92, 93]. Modularity [83] is one of the measures for evaluating communities. Many different methods such as Fast Greedy [94], Walktrap [95], InfoMap [96, 97], Label Propagation [98], Surprise Optimization [99] have been widely used by the researchers to detect communities or clusters in large networks. Fast Greedy algorithm is a hierarchical agglomerative bottom-up clustering approach based on modularity maximization. Walktrap algorithm is based on probability distribution using random walk on graphs. If the random walk is short distanced, then nodes are more likely to remain in the same cluster. InfoMap algorithm is based on information theory principles. Clusters are formed by compressing description of information flows on a network. Label Propagation is an iterative clustering algorithm. Initially each node gets a label. Labels are propagated at each iteration and the algorithm converges when each node has the majority label of its neighbours. Surprise is a global performance metric that uses a cumulative hypergeometric distribution. Optimization of this metric is done using an agglomerative approach. Optimization of modularity is a NP-Complete problem [100] but it is supported by several heuristics [101]. Louvain algorithm [92] is one such heuristic and widely accepted by the researchers. Louvain is much popular for its speed and the output community quality [102]. So, the researchers prefer Louvain over other community detection methods for the parallelization purpose to work with large networks [1, 2, 21, 3].

Due to the advancement of data and computing technologies, graph data is growing at an enormous rate. For example, the number of links in social networks [103, 104, 105] is growing every millisecond. Processing such big graph data requires the development of parallel algorithms [78, 79, 106, 107, 80, 108, 109].
We present a comparative analysis of our shared and distributed memory based parallel Louvain algorithms, their merits and demerits. We have disclosed the problems arisen in communication among processes for distributed memory based parallelism [110, 111]. We also develop a hybrid parallel Louvain algorithm using the advantage of both shared and distributed memory based approaches. The hybrid algorithm gives us the scope to balance between both shared and distributed memory settings depending on available resources. Load balancing is crucial in parallel computing. A straight-forward distribution with an equal number of vertices per processor might not scale well [79]. We also find that load imbalance also contribute to a higher communication overhead for distributed memory algorithms [107]. A dynamic load balancing [106, 112] approach can reduce the idle times of processors leading to increased speedup. Finding a suitable load balancing technique is a challenge in itself as it largely depends on the internal properties of a network and the applications [113]. We also show the performance of MPI (Message Passing Interface) communication for both load imbalanced and load balanced algorithms using MPI profiling tool. The performance analysis shows that the communication overhead for the load imbalanced algorithm is higher than that of the load balanced one.

We present DPLAL (Distributed Parallel Louvain Algorithm with Load-balancing), an efficient algorithm [108, 114] for distributed memory setting, based on a parallel load balancing scheme and graph partitioning. The contributions of this paper are two folds.

- We design several parallel algorithms for Louvain algorithm in shared memory and distributed memory settings. We also design a hybrid parallel algorithm using both shared memory and distributed memory implementations. We present a detailed discussion on the design of the parallel distributed memory Louvain algorithm. This discussion is useful to understand the basic design of the distributed memory parallel algorithm and can be applied by the amateur programmers in High Performance Computing (HPC) for similar problems in multiple domains.

- We show a comparative performance analysis of our parallel algorithms on several real-world networks. We profile our code to find the possible bottlenecks and overcome these hurdles. We also compare the performance of our algorithm with some other prominent algorithms in the literature and get similar or better performance (most cases) in a limited resource environment. This indicates the efficiency and portability of our work to achieve better scalability in highly configured system architectures.

The rest of this paper is organized as follows. We describe the existing related works in Section 2.2. A brief description of the Louvain algorithm, widely used notations throughout the paper and our computational model are presented in Section 2.3. We present the design and complexity analysis of our parallel algorithms.
in Section 2.4. System resources and experimentation datasets are given in Section 2.5. A detailed analysis of the experimental results, comparative analysis with the prominent algorithms in literature and MPI profiling of the code are described in Section 2.6. Section 2.7 concludes the paper with future possibilities.

2.2 Related Works

There exists a rich literature of community detection algorithms [115, 92, 94, 116, 117, 118, 119, 3]. Louvain method [92] is found to be one of the most efficient sequential algorithms [116, 117]. In recent years, several works have been done on the parallelization of Louvain algorithm. Majority of those are shared memory based implementations. Often these implementations either demonstrate only a moderate scalability with limited resources or require highly configured systems for better scalability.

A template has been proposed to parallelize the Louvain method for modularity maximization with a shared memory parallel algorithm [115] using OpenMP (Open Multi-Processing). Maximum modularity has been found by parallel reduction. They have combined communities to supervertices using parallel mergesort. They run their experimental setup on two sets of LFR (Lancichinetti–Fortunato–Radicchi) benchmarks of 8,000 and 10,000 vertices which is a very small number compared to the large network datasets. Another shared memory implementation is done using a hierarchical clustering method with adaptive parallel thread assignment [24]. They have shown that the granularity of threads could be obtained adaptively at run-time based on the information to get the maximal acceleration in the parallelization of Louvain algorithm. They have computed the modularity by adding a neighbor node to the community and assigning some threads in parallel. Dynamic thread assignment of OpenMP has been disabled to let the algorithm adaptively choose the number of threads. For upto 32 cores, the speedup is not significant compared to the previous implementations PLM [120, 119] and CADS [121].

One of the fastest shared memory implementations is Grappolo software package [21, 22], which is able to process a network with 65.6M vertices using 20 compute cores. Authors in [23] claim that their algorithm (Parallel Hybrid Pull-Push Louvain Algorithm) prunes a significant amount of edges and outperforms the fastest shared memory implementation Grappolo [21, 22] by an order of magnitude (3× to 16× faster) with minimal sacrifice to the solution quality. The experiments have been performed on high configuration Intel Skylake machine having 56 cores. Another multi-threaded implementation using pthreads has been done by the authors of [25], particularly designed for fine-grained applications to be executed on large-scale supercomputers. They have used two supercomputers, Power7-IH and Blue Gene/Q with 1,024 nodes (32,768 threads) and 8,192 nodes (524,288 threads) respectively. They could process graph with 138 billion
Another recent work [122] demonstrates that the linear algebraic formulation of the Louvain method can be rapidly implemented using pygraphblas interface. The parallelization has been done using threads. For the largest graph size of around 50K vertices and 1M edges, the speedup for the parallel version is 1.87. The limitation of this work is performance-wise, due to the use of SuiteSparse as a black-box library that is implemented upon the GraphBLAS API. It is unclear how much of the performance is related to the linear algebraic approach or specific implementation decisions. In terms of productivity, the authors keep this implementation ahead of the vertex-based NetworkX implementation. NetworkX requires 158 lines of code (LOC), 4× more compared to the 40 LOC for the pygraphblas implementation. However, the performance is 8× slower with the pygraphblas implementation. The authors claim that the use of the pygraphblas interface will help the graph algorithm developers to simplify the implementation resulting in a boost in algorithmic development productivity.

A GPU implementation, Adaptive CUDA Louvain Method (ACLM) algorithm [123] accelerates the execution time by 45% to 77% compared to the previous shared memory and GPU based parallel algorithms [124, 125, 119] in the large graph benchmarks. This algorithm allocates the optimal number of threads to each block in the GPU and uses the shared memory in GPU blocks to calculate modularity resulting from the merging of two neighbour communities. It calculates the number of required streaming multi-processors (SMs) to allocate threads to each block based on the multiplicand of warps. In this way, the algorithm can reduce the execution time.

A distributed implementation of Louvain algorithm is used to predict communities [126] in social graphs using Logistic Regression Model of Machine Learning [127]. This work focuses on just community prediction rather than detecting communities. Apache Spark’s MLlib is used for implementing this work in a eight node Spark 2.2.0 cluster, with 4 GBs RAM and 1 virtual core per virtual machine. Although the authors show the experimental results with small graphs (36.7K vertices, 0.1M edges), they claim that the performance improvement will certainly be substantially higher in bigger social graphs. The proposed methodology’s complexity is linear and not of a higher polynomial degree as Louvain’s.

One Distributed-Memory implementation has been presented in [3]. Graph partitioning has been done using ParMETIS. They have only parallelized the first level for speedup. Each MPI process locally ignores cross partition edges, that might be an issue with accuracy. They have used three different vertex ordering strategies but there is no significant performance gain. Their approach scales up to 16 – 32 processors but later on the speedup curve flattens for most of the graphs.

Among very few works on distributed memory parallel algorithms, Vite, is a distributed memory im-
plementation of parallel Louvain algorithm [1, 2]. In addition to the baseline parallel implementation of the Louvain algorithm, Vite also includes a number of heuristics that significantly improves performance while preserving the solution quality. Vite has demonstrated speedups in between 2 to 46× using different heuristics for different networks on 4,000 processors. Vite has been run using 12 MPI processes per node and 2 OpenMP threads per process on NERSC Edison, which is a 5,586 node Cray XC30 machine with dual-socket, 12 core Intel Ivy Bridge Xeon E5-2695v2 processor at 2.4 GHz (24 cores per node) [1]. Vite has also been tested on NERSC Cori supercomputer, a 2,388 node Cray XC40 machine with dual-socket Intel Xeon E5 – 2698v3 (Haswell) CPUs at 2.3 GHz per node, 32 cores per node. But the paper uses a higher threshold in lower levels in Louvain method to terminate the level earlier and thus minimized the time contributing to their higher speedup. The work also lacks the emphasis on graph partitioning and balancing load among the processors. This is a clear contrast with our work where we have focused on load balancing issue among others. Our work achieves comparable (or better in many cases) speedups using a significantly fewer number of processors compared to Vite [1, 2].

We have compared our distributed algorithm with Vite. Speedup largely depends upon the system architecture used for the experimentation because MPI performance varies with different system architecture [26]. As similar resource is not available to us during the experimentation, we use our available resource, Louisiana Optical Network Infrastructure (LONI) QB2, which is a cluster containing 504 compute nodes (20 cores per node) with over 10,000 Intel Xeon processing cores of 2.8 GHz. On LONI, full usage of all the processors are not available to the users. So, we could use only 50 computing nodes at a time with 1,000 processors (maximum) for our experiments. We compare our algorithm with the baseline version of Vite, as we have not introduced any heuristics in the basic Louvain algorithm. For the same networks and system architecture, our algorithm achieves better speedup in most of the cases compared to Vite.

2.3 Preliminaries

Notations, different definitions related to the sequential Louvain algorithm, our computational model, and graph partitioning tool, METIS [128] are described in this section.

2.3.1 Notation

The network is denoted by $G(V, E)$, where $V$ and $E$ are the sets of vertices and edges, respectively. Vertices are labeled as $V_0, V_1, \ldots, V_{n-1}$. We use the words node and vertex interchangeably as well as links and edges.
Table 2.1: Terminologies used in explaining the algorithm

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G(V,E)$</td>
<td>graph network with $V$ set of vertices and $E$ set of edges</td>
</tr>
<tr>
<td>$\text{in } [c]$</td>
<td>sum of the weights of the links inside community, $c$</td>
</tr>
<tr>
<td>$\text{tot } [c]$</td>
<td>sum of the weights of the links incident to vertices in community, $c$</td>
</tr>
<tr>
<td>$n2c[i]$</td>
<td>community of vertex, $i$</td>
</tr>
<tr>
<td>$d(i,c)$</td>
<td>number of links from vertex, $i$ to community, $c$</td>
</tr>
<tr>
<td>$v_{sl}$</td>
<td>number of self-loops of vertex, $v$</td>
</tr>
<tr>
<td>$v_{wd}$</td>
<td>weighted-degree of vertex, $v$</td>
</tr>
<tr>
<td>$n =</td>
<td>V</td>
</tr>
<tr>
<td>$v_{SN}$</td>
<td>1st vertex of a processor</td>
</tr>
<tr>
<td>$V_u$</td>
<td>set of vertices requiring update</td>
</tr>
<tr>
<td>$W$</td>
<td>set of all processors</td>
</tr>
<tr>
<td>$N =</td>
<td>W</td>
</tr>
<tr>
<td>$P_s$</td>
<td>set of processors where message will be sent</td>
</tr>
<tr>
<td>$P_r$</td>
<td>set of processors where message will be received</td>
</tr>
<tr>
<td>$M$</td>
<td>message</td>
</tr>
</tbody>
</table>

$P$ is the number of processors used in the computation, denoted by $P_0, P_1, \ldots, P_{N-1}$ where $0, 1, 2, \ldots, N-1$ refers to the rank of a processor. Terms frequently used throughout the paper are enlisted in Table 2.1.

We also define some operators those we use in Section 2.4 to describe our algorithms. Our defined operators and their usage is described in Table 2.2.

Table 2.2: User-defined operators used in the algorithm

<table>
<thead>
<tr>
<th>Operator</th>
<th>Use</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\sqsubset)</td>
<td>$a \sqsubset P_b$</td>
<td>Vertex $a$ is partitioned under Processor $P_b$</td>
</tr>
<tr>
<td>(&lt;)</td>
<td>$x &lt; y$</td>
<td>Message $x$ contains $y$</td>
</tr>
<tr>
<td>(\rightarrow)</td>
<td>$P_a \xrightarrow{M_{a,b}} P_b$</td>
<td>Processor $P_a$ is sending message, $M_{a,b}$ to processor $P_b$</td>
</tr>
</tbody>
</table>

2.3.2 Louvain Algorithm for Community Detection

Louvain is a simple heuristic method to extract the community structure of large networks based on modularity optimization [92]. It outperforms all other known community detection methods in terms of computation time.

Modularity, $Q$ is calculated using Equation 2.1, where $-1 < Q < 1$.

$$Q = \frac{1}{2m} \sum_{i,j} \left[ A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i,c_j)$$ (2.1)

The meaning of the symbols is described in Table 2.3. The algorithm is divided into two phases which
Table 2.3: Symbols used for calculating Modularity in Equation 2.1 and Equation 2.2

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>Modularity</td>
</tr>
<tr>
<td>A</td>
<td>Usual adjacency matrix</td>
</tr>
<tr>
<td>$A_{ij}$</td>
<td>Link weight between nodes i and j</td>
</tr>
<tr>
<td>m</td>
<td>Total link weight in the network</td>
</tr>
<tr>
<td>$k_i$</td>
<td>Sum of the link weights attached to node i</td>
</tr>
<tr>
<td>$\frac{k_i}{2m}$</td>
<td>Average fraction of weight that would be assigned to node j, if node i assigned its link weight randomly to other nodes in proportion to their own link weights</td>
</tr>
<tr>
<td>$A_{ij} - \frac{k_i k_j}{2m}$</td>
<td>How strongly nodes i and j are connected in the real network, compared to how strongly connected we would expect them to be in a random network</td>
</tr>
<tr>
<td>$c_i$</td>
<td>Community to which node i is assigned</td>
</tr>
<tr>
<td>$\delta(c_i, c_j)$</td>
<td>Kronecker delta. Value is 1 when nodes i and j are assigned to the same community. Otherwise, the value is 0</td>
</tr>
<tr>
<td>$\Delta Q$</td>
<td>Gain in Modularity</td>
</tr>
<tr>
<td>$\sum_{in}$</td>
<td>Sum of the weights of the links inside community C</td>
</tr>
<tr>
<td>$\sum_{tot}$</td>
<td>Sum of the weights of the links incident to nodes in community C</td>
</tr>
<tr>
<td>$k_{i,in}$</td>
<td>sum of the weights of the links from node i to nodes in C</td>
</tr>
</tbody>
</table>

are iteratively repeated.

2.3.2.1 Modularity Optimization Phase

For each node $i$, all the neighbours $j$ belonging to the neighborhood of $i$ are considered for computing the modularity value. The gain in modularity, $\Delta Q$ is evaluated by removing $i$ from its community and by placing it in the community of $j$. $\Delta Q$ obtained by moving an isolated node $i$ into a community $C$ is computed using Equation 2.2.

$$\Delta Q = \left[\frac{\sum_{in} + k_{i,in}}{2m} - \left(\frac{\sum_{tot} + k_i}{2m}\right)^2\right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right]$$ (2.2)

The symbolic meanings are given in Table 2.3. When $i$ is removed from its community $C$ a similar equation is used. This continues repeatedly until no further improvement is achieved. This phase comes to an end when a local maxima of the modularity is attained.

2.3.2.2 Community Aggregation Phase

In this phase, a new network is formed using the communities found during the first phase as nodes. The weights of the links between two new nodes are given by the sum of the weight of the links between nodes
in the corresponding two communities. Links between nodes of the same community lead to self-loops for
the community in the new network.

After completion of second phase, the first phase of the algorithm is reapplied to the resulting weighted
network and iteration continues as long as positive gain in modularity value is achieved.

2.3.3 Computational Model

At first, we develop a shared memory based parallel algorithm using Open Multi-Processing (OpenMP) to
eliminate the limitations of [115] with a different approach. Shared memory based algorithms have some
own limitations due to limited number of physical processing cores hindering high speedup. Therefore, we
develop MPI based parallel algorithm for distributed memory systems, where each processor has its own
local memory. The processors do not have any shared memory, one processor cannot directly access the local
memory of another processor, and the processors communicate via exchanging messages using MPI.

2.3.4 Graph Partitioner, METIS

METIS [128] is an open source tool for partitioning irregular large graphs using multilevel graph partitioning
paradigm. For traditional graph partitioning problem, the objective is to minimize the number of edges
(unweighted graphs) or sum of weights (weighted graphs) while computing the k-way partitioning. Along
with this, METIS also supports to directly minimize the total communication volume resulting from the
partitioning.

2.4 Methodology

In this section we describe the design of different parallel implementations of the Louvain algorithm and
analyze the runtime complexity of the algorithms.

2.4.1 Shared Memory Parallel Louvain Algorithm

In shared memory based algorithms, there is a shared address space and multiple threads share this common
address space. This shared address space can be used efficiently using lock and other synchronization
techniques. The main hindrance behind the shared memory based systems is the limited number of processing
cores. We parallelize the Louvain algorithm by distributing the computational task among multiple threads
using Open Multi-Processing (OpenMP) framework. We parallelize the Louvain algorithm computational
task-wise in a straight-forward approach. Whenever there is a need to iterate over the full network or even
the neighbors of a node, considering the large network size, the work is done by multiple threads to minimize the workload and do the computation faster.

2.4.2 Distributed Memory Parallel Louvain Algorithm

Our aim is to compute the communities of the full network in a distributed manner. To serve the purpose, we distribute the full network among the processors in a balanced way, so that each processor can do its computation in a reasonable time and no processor should wait for another processor. It is necessary because after each level of computation, the processors have to communicate with the root processor to generate the modularity of the full network. After each level of iteration, the network size decreases gradually and when the network size is considerably small, the final level is computed by a single processor which acts similar to the Louvain sequential algorithm.

2.4.2.1 Graph Partitioning

Let $N$ be the number of processors used in the computation. The network is divided into $N$ partitions, and each processor, $P_i$ is assigned one such partition $G_i(V_i, E_i)$. Processor $P_i$ performs computation on its partition $G_i$. The network data is given as input in a single disk file. While partitioning $G(V, E)$, if the vertex-id does not start with 0, then the vertices are renumbered from 0 to $n_i - 1$. The network is partitioned depending on the number of vertices of the network such that each processor gets equal number of vertices. $N$ can vary depending on the system configuration and available resources. $n$ is also of varied range depending on the number of vertices in the network. So, we cannot divide the vertices equally among processors when $(n \mod N) \neq 0$ We distribute the remaining nodes starting with processor $P_0$ and continue up to processor $P_k (0 \leq k < N)$, as long as the remainder lasts. Each processor has its own part of the network necessary to compute the communities in the partial network. It is a very naive partitioning technique. Initially we apply this technique because in many cases, the naive way often works better.

2.4.2.2 Community Detection

We have parallelized the sequential algorithm in such a way that each processor can compute its partial network’s community with minimized communication among the processors. The following information are needed for each processor to complete its part of computation:

- Degree of each vertex within the partition
- Neighbor list of each vertex
• Weight associated with each neighbor

**Algorithm 1: Our Naive Distributed Parallel Louvain using MPI**

---

**Data:** Input Graph $G(V,E)$

**Result:** (Vertex, Community) Pair

1. **while increase in modularity do**
2.   G $(V,E)$ is divided into $p$ processes;
3.   Each graph $_i.bin$ contains $\left\lceil \frac{n}{p} \right\rceil$ vertices and corresponding edges in adjacency list format;
4. **for Each processor $P_i$ (executing in parallel) do**
5.   Initialize_Graph();
6.   Exchange_Starting_Node();
7.   Gather Neighbour Info();
8.   Compute_Community();
9.   Exchange Updated_Community();
10.  Resolve Community Duality();
11.  Exchange Duality Resolved Community();
12.  Find Unique Communities();
13.  Compute Modularity();
14.  **if root processor then**
15.     Generate NextLevel Graph();
16.  **end**
17. **end**
18. **if number of nodes in new graph < $i$ then**
19.     $i \leftarrow \frac{\text{number of nodes in new graph}}{2}$
20. **end**
21. **end**

---

In first phase, each processor scans through all neighboring vertices and identifies those in different processors. It then gathers the mentioned information by message passing among those identified processors. Then each processor locally computes the modularity of the partial network and does community detection. After computation, each processor sends information of each vertex’s community to the processor acting as the root. The root processor needs the value of $in$, $tot$ arrays and total weight of the network to compute modularity of the full network as given in Equation 2.2. This full process is iterated several times as long as there is increase in modularity.

The output is stored as (node,community) for each level of iteration. It is also stored in an adjacency matrix format. We have used the matrix format to visualize the graph using Python’s **NetworkX** library.

Algorithm 1 represents the pseudo-code of our approach. We have described all the steps (1 – 11) below that is given in Lines 5 – 15 of the pseudo-code. These steps are done for each level of computation and repeated as long as we get increase in the modularity value.
2.4.2.2.1 **Step 1: Graph Initialization**  In this step, each processor \( P_i \) reads the input graph, \( graph\_name\_i.bin \). The nodes are renumbered if necessary. Afterwards, for graph \( g \), we initialize the arrays \( tot \), \( in \) and \( n2c \). Line 5 in Algorithm 1 denotes this step.

2.4.2.2.2 **Step 2: Exchange of Starting Node**  In this step, each processor \( P_i \) sends own starting node-id to other processors, \( P_j \) in the communicating world.

\[
P_i \xrightarrow{M_{i,j}} P_j
\]

where

\[
i, j \in W, \; j = 0, 1, \ldots, N - 1 \quad \text{and} \quad i \neq j
\]

\[
M_{i,j} \subseteq V_{SN}
\]

\[
V_{SN} \subseteq P_i
\]

The number of Send and Receive operations are equal to \( N - 1 \). This step is mainly required because at later phases, we use this information to find out which node belongs to which processor at other steps of our calculation. If number of processors is always a factor of network size, we could have skipped this step by numerically figuring out the starting node for each processor. This is a very rare scenario and our world size can vary depending on the system configuration and available resources. Again, network size is also of varied range. So, we cannot divide the nodes equally among processors when \((n \; \text{mod} \; N) \neq 0\). So, we distribute the remaining nodes starting with processor \( P_0 \) and continue up to processor \( P_k \), as long as the remainder lasts. Here, \( 0 \leq k < N \).

2.4.2.2.3 **Step 3: Collection of Neighbour Information**  In this step, we scan through the neighbor list of all vertices and find out the neighbors those do not belong to current processor using the information of Step 2.

Now, we need the following information of each vertex in the neighbor list: degree, weight, and neighbor list with weights.

These are necessary for the calculation in Step 4. To gather this information, we need to know beforehand, how many send and receive operations are required to communicate with other processors. To calculate the number of sending and receiving processors, we iterate over the full neighbor list of all vertices. Thus we
Algorithm 2: Gather_Neighbour_Info()

1. Initialize empty array receive_list;
2. for each vertex v in g do
   3. Initialize empty array send_list;
      4. for each neighbor of v do
         5. $V_n \leftarrow$ neighboring node of v;
         6. if $V_n$ not in current processor then
            7. $P_j \leftarrow$ processor having $V_n$;
            8. if receive_list.contains() $\neq V_n$ then
               9. receive_list.push($V_n$);
            10. end
            11. if send_list.contains() $\neq P_j$ then
                12. send_list.push($P_j$);
            13. end
         14. end
      15. end
   16. for times=send_list.size() do
      17. $P_j \leftarrow$ send_list.pop();
      18. MPI_Isend(deg(v), weight(v), $P_j$);
      19. MPI_Isend (v, n2c[v], neighbors of v (node-id, weight), $P_j$);
   20. end
   21. for times=receive_list.size() do
      22. MPI__Recv(deg(v'), weight(v'), MPI_ANY_SOURCE);
      23. $P_j \leftarrow$ status.MPI_ANY_SOURCE ;
      24. MPI_Recv (v', n2c[v'], neighbors of v' (node-id, weight), $P_j$);
      25. Update n2c[v'], in[v'] and tot[v'] locally;
      26. Neighbor[v']. push (neighbors of v' (node-id, weight));
   27. end

Figure 2.1: A Sample Graph to explain the algorithmic steps used in Section 2.4.2.
find out the vertices those do not belong to current processor. We also identify the unique processors those we need to communicate in later steps.

The number of unique processors is the size of our send-list. The number of unique vertices found from neighbor list of all the vertices, is the size of our receive-list. After receiving the desired information, we update the \( n2c \), \( in \) and \( tot \) array entries for the aforementioned vertices and store the neighbor list with weight information for further calculation.

Pseudo-code in Algorithm 2 represents this step. Figure 2.1 visualizes a network of 16 nodes that helps us to understand the process of the current and further steps of the algorithm. Table 2.4 shows an example simulation for this step.

Table 2.4: Example Illustration for Send List and Receive List Count given in Step 3 [Algorithm 2]

<table>
<thead>
<tr>
<th>Processor</th>
<th>Node-Id</th>
<th>Neighbor-List</th>
<th>Send-List (Processor)</th>
<th>Receive-List (Vertex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>2,3,4,5</td>
<td>1 (P1)</td>
<td>4 (4, 5, 7, 6)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2,4,7</td>
<td>1 (P1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0,1,4,5,6</td>
<td>1 (P1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0,7</td>
<td>1 (P1)</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>0,1,2,10</td>
<td>2 (P0, P2)</td>
<td>6 (0, 1, 2, 10, 11, 3)</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0,2,7,11</td>
<td>2 (P0, P2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>2,7,11</td>
<td>2 (P0, P2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>1,3,5,6</td>
<td>1 (P0)</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>8</td>
<td>9,10,11,14,15</td>
<td>1 (P3)</td>
<td>7 (14, 15, 12, 4, 13, 5, 6)</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>8,12,14</td>
<td>1 (P3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>4,8,11,12,13,14</td>
<td>2 (P1, P3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>5,6,8,10,13</td>
<td>2 (P1, P3)</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>12</td>
<td>9,10</td>
<td>1 (P2)</td>
<td>4 (9, 10, 11, 8)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>10,11</td>
<td>1 (P2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>8,9,10</td>
<td>1 (P2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>8</td>
<td>1 (P2)</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>21</strong></td>
<td></td>
<td><strong>21</strong></td>
</tr>
</tbody>
</table>

2.4.2.4 Step 4: Community Computation  In this step, we do the actual computation to determine community of each vertex. Each processor, \( P_i \), completes this step individually and locally updates the community of the vertices belonging to it. This step does not require any communication among processors. A random vertex, \( V \) is chosen from the list of vertices. Then, the set of neighboring communities of that vertex is computed with the information from Step 3.

Here, the number of links from that vertex, \( V \) to all its neighboring community is computed and stored. Next, the vertex is removed from its current community, \( C_{old} \). Now, the modularity gain is calculated for all its neighboring communities. If the gain is maximum for community, \( C \), then vertex, \( V \)'s community
Algorithm 3: Compute_Community()

1 while vertices.size()! = 0 do
2     Pick random vertex, V from g ;
3     for all neighboring vertices of V do
4         $V_n \leftarrow$ neighboring vertex of V ;
5         $d_{V,V_n} \leftarrow$ number of links from vertex, V to community, $V_n$;
6     end
/* remove node from its current community */
7     remove (V, n2c[V], d_{V,V_n});
8     for all neighboring vertices of V do
9         $V_n \leftarrow$ neighboring vertex of V ;
10        compute_gain() ;
11        if positive gain then
12            n2c[v] $\leftarrow$ n2c[V_n] ;
13            if n2c[V_n] not in current processor then
14                P_j $\leftarrow$ processorhaving n2c[V_n];
15                rem_comm[V_n] $\leftarrow$ n2c[V_n];
16                rem_dvc[V_n] $\leftarrow$ d_{V,V_n};
17            end
/* insert locally */
18        Insert (V, n2c[V_n], d_{V,V_n});
19     end
20 end
21 end

is updated to $C$. Otherwise, V gets back to its previous community, $C_{old}$. This, remove () and insert () operations update the in and tot arrays implicitly. As, each processor is locally updating the community, we need to keep trace if community $C$ belongs to processor $P_i$ or not. If $C$ belongs to processor, $P_j$, we simply store $V_{comm} = C$ and $V_{link} = d_{V,C}$ where $d_{V,C} =$ number of links from vertex $V$ to Community $C$. These two values $V_{comm}$ and $V_{link}$ are stored separately in two arrays rem_comm and rem_dvc that we have used later in Step 6. This continues until all vertices from the vertex-list of the processor is covered. The pseudo-code is given in Algorithm 3.

2.4.2.2.5 Step 5: Update of Processor List for Communication This is a preliminary step required for Step 6. After updating the community of the vertices locally, we need to circulate the update globally among all processors so that each community keeps aware of the vertices in its region. Again, we need to figure out the number of sending and receiving processors before starting the communication. It is done in this step. We iterate over all the elements of rem_comm array and find out the processor to which that element belongs to. For each processor, we sum the total number of vertices those require update. Again, we also send to all processors of the world a message, msg containing either 0 or 1 and receiving back
from all. Now, the processors those have at least one vertex that requires update, are inserted into our list of sending processors and the value $msg = 1$. Otherwise, $msg = 0$. So, the receive list size is computed by summing the received value of $msg$. Algorithm 4 represents the pseudo-code.

$$P_i \xrightarrow{M_{i,j}} P_j$$

where,

$$i,j \in W, \ j = 0,1,\ldots,N-1 \cap i \neq j$$

$$M_{i,j} \begin{cases} 1, |V_u| > 0 \\ 0, |V_u| = 0 \end{cases}$$

2.4.2.2.6 Step 6: Exchange of Updated Community  In this step, the update of community is done globally among all processors. Now, Processor, $P_i$ sends the receiving processor, $P_j$ the following data until the send-list becomes empty:

- vertex $v$ belonging to Processor $P_i$, whose community is in Processor $P_j$,
- vertex $v$’s community $C$ belonging to Processor $P_j$,
- number of links from vertex $v$ to community $C$,
- number of self-loops of vertex $v$, and
- weighted degree of vertex $v$. 

---

**Algorithm 4: Update_Send_Receive_List()**

1. for times = world_size do
2.     if $P_i \neq rank$ then
3.         if $|V_u| > 0$ then
4.             MPI_Send (1, $P_i$);
5.             send_list.push($P_i$);
6.         else
7.             MPI_Send (0, $P_i$);
8.         end
9.     MPI_Recv (j, $P_i$);
10.    rcv_list_size+ = j;
11. end
12. end
\[ P_i \xrightarrow{M_{i,j}} P_j \quad \text{where, } j \in P_s \]

\[ M_{i,j} \propto v, n2c[v], d(v, n2c[v]), v_{sl}, v_{wd} \]

\[ v \equiv P_i, n2c[v] \equiv P_j \]

Upon receiving the data, the vertex is inserted to its community and it updates \textit{in} and \textit{tot} arrays internally. Pseudo-code for this step is given in Algorithm 5.

**Algorithm 5:** Exchange Updated Community()

1. \textbf{for} \( \text{times} = \text{send\_list.size()} \) \textbf{do}
2. \hspace{1em} \text{MPI\_Send} (#nodes, \( P_j \));
3. \hspace{1em} \text{MPI\_Send} (v, n2c[V_n], dnodecomm, \( v_{sl}, v_{wd} \), \( P_j \));
4. \textbf{end}
5. \textbf{for} \( \text{times} = \text{rcv\_list\_size} \) \textbf{do}
6. \hspace{1em} \text{MPI\_Recv} (# nodes, MPI\_ANY\_SOURCE);
7. \hspace{1em} \text{P}_i, \text{status.MPI\_ANY\_SOURCE};
8. \hspace{1em} \text{MPI\_IRecv} (x, y, dnodecomm_{xy}, x_{sl}, x_{wd}, P_i);
9. \hspace{2em} /* insert x in community y global update, only update \textit{in[]}, \textit{tot[]} of y */
10. \hspace{1em} \text{Insert} (x, y, dnodecomm_{xy}, x_{sl}, x_{wd});
11. \textbf{end}

**2.4.2.2.7 Step 7: Resolving Community Duality** After the previous step, there remains an inconsistency to calculate total number of communities in the full network. The same network shown in Figure 2.1 is used to demonstrate the problem. Basically, Vertex 1 and Vertex 4 belong to the same community. The community label can be either 1 or 4. The community label does not have any effect on the result. But in current scenario, same community will be counted twice. So, we need to eliminate this problem. To solve this, we reserve the communities with higher labels and change the lower labels to higher ones. So now, in Table 2.5, vertex 4’s community is changed again from community 1 to community 4 because 4 is larger than 1. Vertex 1 retains its community 4. As vertex 4 belongs to Processor 1, Processor 0 needs to communicate with it to circulate the update.

Next, we update the vertices’ communities those belong to current processor and track the communities those belong to different processors. Therefore, communication is required. So, for other processors those contain communities of the vertices, we sum the total number where current community label is less than the vertex-id. We store this information in an array, \textit{count\_community} and it will be used in Step 8. We store the vertices whose community will be updated after the communication. We also enlist the unique community labels whose updated community we will receive upon completion of the MPI communication.
Table 2.5: Example Illustration for Community Duality Problem given in Step 7

<table>
<thead>
<tr>
<th>Processor</th>
<th>Vertex</th>
<th>Community</th>
<th>Problem Scenario</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>1</td>
<td>4</td>
<td>Vertex 1 switches its community to vertex 4</td>
<td>Vertex 1 retains community 4</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>1</td>
<td>Vertex 4 switches its community to vertex 1</td>
<td>Vertex 4’s community is switched back to vertex 4</td>
</tr>
</tbody>
</table>

These are required for Step 9.

2.4.2.2.8 Step 8: Update of Processor List for Communication in Resolving Community Duality

It is a preliminary step required for Step 9. This step is similar to the process as described in Step 5. Here, we iterate over the array, count_community and store the indices i in send_list if count_community[i] > 0. i represents the processor number for sending. Update of receive list is also analogous to the calculation as described in Step 5 for finding out the size of receive_list.

Algorithm 6: Resolve_Community_Duality() and Exchange_Duality_Resolved_Community()

```python
1 for each vertex, v do
2   if n2c[v] < v then
3     if n2c[v] not in P_i then
4       P_j ← processor having n2c[v];
5       MPI_Send(n2c[v], P_j);
6       MPI_Recv (n2c[v], n2c[n2c[v]], P_j);
7       n2c[v] ← n2c[n2c[v]];  
8     else
9       comm ← n2c[v];
10      n2c[v] ← n2c[comm];
11   end
12 end
```

2.4.2.2.9 Step 9: Exchange of Duality Resolved Community

In this step, processor P_i sends the community label to all processors P_j belonging to P_i’s send-list to receive the updated community of that particular community. P_j after receiving the message, sends back the corresponding community of the received id to P_i. After receiving info from P_j, P_i updates the community of the vertices those need updates, using the values from Step 7. In this communication step, subsequent send-receive is done.
\[ P_i \xrightarrow{M_{1,i,j}} P_j \quad \text{where, } j \in P_s \]

\[ M_{1,i,j} \triangleq C; C \subseteq P_j \]

where, \( C \) denotes the vertex whose community label needs to be updated.

\[ P_j \xrightarrow{M_{2,j,i}} P_i \]

\[ M_{2,j,i} \triangleq C; n2c(C); C \subseteq P_j \]

A brief pseudo-code to represent steps 7 and 9 is given in Algorithm 6.

2.4.2.2.10 Step 10: *Finding Unique Communities and Computation of Modularity* In this step, each processor \( P_i \), where \( 0 \leq i < N \) finds out all the unique communities by iterating over all of its vertices belonging to it. Pseudo-code is presented in Algorithm 7.

**Algorithm 7: Find_Unique_Communities()**

1. Initialize empty array unique_community;
2. for each vertex, \( v \) do
3. \[ \text{comm} \leftarrow n2c[v]; \]
4. \[ \text{if unique_community.contains()}! = \text{comm} \text{ then} \]
5. \[ \text{unique_community.push(comm);} \]
6. end
7. end

To calculate total unique communities in the world, each processor \( P_k \), where \( 1 \leq k < N \) sends its unique_community list to root processor \( P_0 \). \( P_0 \) then merges all the unique communities received from each processor \( P_k \) and eliminates the duplicate ones.

**Algorithm 8: Compute_Modularity()**

1. if \( \text{rank}! = 0 \) then
2. \[ i \leftarrow \text{rank}; \]
3. \[ m \leftarrow \text{total_weight}; \]
4. \[ \text{in1} \leftarrow (\sum_{\text{node}} \text{in}) ; \]
5. \[ \text{tot1} \leftarrow (\sum_{\text{node}} \text{m})^2 ; \]
6. \[ \text{MPI_Send(community array, in1, tot1, 0);} \]
7. else
8. \[ \text{MPI_Recv (community array, in1, tot1, MPI_ANY_SOURCE);} \]
9. \[ \text{Modularity} \leftarrow \sum \text{in1} - \sum \text{tot1;} \]
10. end
Along with the unique_community list, $P_k$ also sends to $P_0$ values $in1$ and $tot1$ calculating the values from $in$ and $tot$ arrays. These two values are required for calculation of modularity of the full network. Algorithm 8 represents the pseudo-code for this step.

2.4.2.2.11 Step 11: Generating Next Level Graph This step is performed by only the root processor $P_0$. $P_0$ renumbers the communities from 0 to $Z − 1$ to formulate the new input graph to be used for next level.

$$Z = \text{Total number of unique communities after merging and eliminating duplicate}$$

So, $Z$ is the number of vertices for input graph of next level. The connectivity of edges and corresponding weights are calculated from available data and the new graph is generated. Steps 1 to 11 are done for each level of computation and repeated as long as we get increase in modularity value.

2.4.2.2.12 Complexity Analysis We analyze the run-time complexity of our distributed memory MPI based Algorithm 1. Suppose, $n_i$ be the number of vertices in processor $p_i$ and $N$ be the total number of processors. For any vertex in the processor, $x$ the degree of vertex $x$ is denoted by $\text{deg}(x)$. In Line 5, each processor initializes some arrays of size $n_i$. So, the computational complexity is $O(n_i)$. In Line 6, from Step 2 we find that the communication complexity is $O(N − 1)$. In Line 7, from Algorithm 2: Lines 4−15, depending on the for loop, we get that the computational complexity becomes $O(n_i \sum_{m=1}^{n_i} \text{deg}(x_m))$. Regarding communication complexity it would be $O(N − 1)$ in the worst case if the neighbors of a vertex are scattered in all of the processors. For Algorithm 3, the computational complexity is $O(n_i \sum_{m=1}^{n_i} \text{deg}(x_m))$. For Algorithm 4, the communication complexity is $O(N − 1)$. For Algorithm 5, the only computation part is the Insert function in line 9. For insertion or update operation in array the computational complexity is $O(1)$. The communication complexity is $O(N − 1)$. For Algorithm 6, the computational complexity is $O(n_i)$ and the communication complexity is $O(N − 1)$. For Algorithm 7, the computational complexity is $O(n_i)$. For Algorithm 7, the communication complexity is $O(N − 1)$. So, considering all of the steps in Algorithm 1, the total complexity of Algorithm 1 is given in Equation 2.3. From Equation 2.3, we see that there is only a single quadratic term that is dependent upon the degree distribution of the vertices of the entire network. So, in case of sparse graphs, the complexity of the algorithm would be linear time.
\[
O\left(\frac{1}{N}[n_i + (N - 1) + n_i \sum_{m=1}^{n_i} \deg(x_m) + (N - 1) + 
\sum_{m=1}^{n_i} \deg(x_m) + 1 + (N - 1) + n_i + N - 1 + n_i + N - 1]\right)
\]

\[
= O\left(\frac{1}{N}[\alpha N + \beta n_i \sum_{m=1}^{n_i} \deg(x_m)]\right)
\]

where, \(\alpha, \beta =\) Multiplying Factors

### 2.4.3 Hybrid Parallel Louvain Algorithm

We use both MPI and OpenMP together to implement the Hybrid Parallel Louvain Algorithm. The hybrid version gives us the flexibility to balance between both shared and distributed memory system. We can tune between shared and distributed memory depending on available resources. In the multi-threading environment, a single thread works for communication among processors and other threads do the computation.

### 2.4.4 Distributed Parallel Louvain Algorithm with Load-balancing (DPLAL)

To implement DPLAL, we use the similar approach as described in Section 2.4.2. In the first phase, we have used well-known graph-partitioner METIS [128] to partition our input graph to distribute among the processors. Depending on METIS output, we adjust the number of processors because METIS does not always create same number of partitions as provided in input. We use both edge-cut and communication volume minimization approaches. An empirical comparison of these approaches is described later in Section 2.6.

After partitioning, we distribute the input graph among the processors. For second phase, we follow the same flow as described in the Algorithm 1. But we have to recompute each function that has been calculated from input graph. Runtime analysis for each of these functions being used in MPI communication has been demonstrated in Section 2.6. Our incorporation of graph partitioning scheme helps minimize the communication overhead of MPI to a great extent and we get an optimized performance from DPLAL.

**Complexity Analysis** For Algorithm 9, in line 2, the number of partitions is dependent upon the minimum number of cut-edges or minimization of communication volume. So, the total number of processors, \(N\) is a factor of cut-edges \(\overrightarrow{E}\) or communication volume \(V\). So we can write the Equation 2.4 or Equation 2.5 depending on the METIS option.

\[
N = \gamma \overrightarrow{E}
\]
Algorithm 9: DPLAL-Distributed Parallel Louvain Algorithm using Load-balancing

Data: Input Graph G(V,E) [Edge List Format]
Result: (Vertex, Community) Pair

while increase in modularity do
  G (V, E) is partitioned using METIS into p' partitions;
  G'(V,E) pre-processed according to METIS-Output;
  G"(V,E) converted to Adjacency List format to be given to each processor ;
  for Each processor $P_i$ (executing in parallel) do
    Initialize_Graph();
    Exchange_Starting_Node();
    Gather_Neighbour_Info();
    Compute_Community();
    Exchange_Updated_Community();
    Resolve_Community_Duality();
    Exchange_Duality Resolved_Community();
    Find_Unique_Communities();
    Compute_Modularity();
    if root processor then
      Generate_NextLevel_Graph();
    end
  end
  if number of nodes in new graph < i then
    $i \leftarrow \frac{\text{number of nodes in new graph}}{2}$
  end
end

$$N = \gamma V$$ (2.5)

The rest of the complexity of Algorithm 9 is computed similarly as Algorithm 1. For the total complexity analysis, we can replace $N$ given in Equation 2.3 by Equation 2.4 or 2.5 depending on the METIS option. So, total complexity for Algorithm 9 is given in Equation 2.6.

$$O\left(\frac{1}{\gamma^z}[\alpha \gamma z + \beta n_i \sum_{m=1}^{n_i} \deg(x_m)]\right)$$

where,

$$z = E||V$$

$$\alpha, \beta, \gamma = \text{Multiplying Factors}$$
2.5 Experimental Setup

We describe our experimental setup and datasets in this section. We use large-scale compute cluster for working on large real-world graph datasets.

2.5.1 Execution Environment

We use Louisiana Optical Network Infrastructure (LONI) QB2 [129] compute cluster to perform all the experiments. QB2 is a 1.5 Petaflop peak performance cluster containing 504 compute nodes with over 10,000 Intel Xeon processing cores of 2.8 GHz. We use at most 50 computing nodes with 1000 processors for our experiments. We use TAU (Tuning and Analysis Utilities) [130] toolkit for profiling our parallel programs written in C++.

2.5.2 Description of Datasets

We have used real-world networks from SNAP [131] depicted in Table 2.6. We have performed our experimentation on different domains of network including social networks, internet, peer-to-peer networks, road networks, network with ground truth communities, and Wikipedia networks. All these networks show different structural and organizational properties. This gives us an opportunity to assess the performance of our algorithms for worst case inputs as well. The size of graphs used in our experiments ranges from several hundred thousands to millions of vertices and edges.

To illustrate a simple example from our dataset, we choose soc-LiveJournal1, a social network [131] that is generated from the users of a popular blogging site LiveJournal. So this network consists of all user-to-user links. Applying our community detection algorithm to this input network, we can figure out the social groups formed with a number of users. This group formation can be based on their similar interest to journals or blog topics.

2.6 Result

We present the scalability and runtime analysis of our algorithms in this section. We have calculated the speedup as the ratio of the single thread/processor time and the multi thread/processor time. We discuss the trade-offs and challenges alongside. We present a comparative analysis with other prominent algorithms in literature. We profile our code with MPI Profiling tool TAU and include the analysis as well.
Table 2.6: Datasets used in our experimental evaluation

<table>
<thead>
<tr>
<th>Network</th>
<th>Vertices</th>
<th>Edges</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>email-Eu-core</td>
<td>1,005</td>
<td>25,571</td>
<td>Email network from a large European research institution</td>
</tr>
<tr>
<td>ego-Facebook</td>
<td>4,039</td>
<td>88,234</td>
<td>Social circles ('friends lists') from Facebook</td>
</tr>
<tr>
<td>wiki-Vote</td>
<td>7,115</td>
<td>103,689</td>
<td>Wikipedia who-votes-on-whom network</td>
</tr>
<tr>
<td>p2p-Gnutella08</td>
<td>6,301</td>
<td>20,777</td>
<td>A sequence of snapshots of the Gnutella network</td>
</tr>
<tr>
<td>p2p-Gnutella09</td>
<td>8,114</td>
<td>26,013</td>
<td>for different dates of August 2002</td>
</tr>
<tr>
<td>p2p-Gnutella04</td>
<td>10,876</td>
<td>39,994</td>
<td></td>
</tr>
<tr>
<td>p2p-Gnutella25</td>
<td>22,687</td>
<td>54,705</td>
<td></td>
</tr>
<tr>
<td>p2p-Gnutella30</td>
<td>36,682</td>
<td>88,328</td>
<td></td>
</tr>
<tr>
<td>p2p-Gnutella31</td>
<td>62,586</td>
<td>147,892</td>
<td></td>
</tr>
<tr>
<td>soc-Slashdot0922</td>
<td>82,168</td>
<td>948,464</td>
<td>Slashdot social network from February 2009</td>
</tr>
<tr>
<td>com-DBLP</td>
<td>317,080</td>
<td>1,049,866</td>
<td>DBLP collaboration (co-authorship) network</td>
</tr>
<tr>
<td>roadNet-PA</td>
<td>1,088,092</td>
<td>1,541,898</td>
<td>Pennsylvania road network</td>
</tr>
<tr>
<td>roadNet-CA</td>
<td>1,965,206</td>
<td>2,766,607</td>
<td>California road network</td>
</tr>
<tr>
<td>com-Orkut</td>
<td>3,072,441</td>
<td>117,185,083</td>
<td>Orkut online social network</td>
</tr>
<tr>
<td>soc-LiveJournal1</td>
<td>4,847,571</td>
<td>68,993,773</td>
<td>LiveJournal online social network</td>
</tr>
</tbody>
</table>

2.6.1 Speedup factors of shared and distributed memory algorithms

We design both shared and distributed memory based algorithms for Louvain methods. The speedup results are shown in Fig. 2.2a and Fig. 2.2b. Our shared memory and distributed memory algorithms achieve speedups of around 4 and 1.5 respectively. The number of physical processing cores available to our system is 20. Our shared memory algorithm scales well to this many cores. However, due to the unavailability of large shared memory system, we design distributed memory algorithm.

Figure 2.2: Speedup factors of our parallel Louvain algorithms for different types of networks. Our hybrid algorithm strikes a balance between shared and distributed memory based algorithms.

Further, shared memory algorithms show a limited scalability to large networks as discussed in [115]. Our
distributed memory algorithm demonstrates only a minimal speedup for 30 processors. The inter-processor communication severely affects the speedup of this algorithm. We strive to overcome such communication bottleneck by designing hybrid algorithm.

Our hybrid algorithm tends to find a balance between the above two approaches, shared and distributed memory. As shown in Fig. 2.2c, we get a speedup of around 2 for the hybrid implementation of Louvain algorithm. The speedup is similar to the MPI implementation. It is evident that in multi-threading environment runtime will decrease as workload is distributed among the threads. But we observe that in some cases, both single and multiple threads take similar time. Even sometimes multiple threads take more time than a single thread. It indicates that hybrid implementation also suffers from the communication overhead problem alike MPI implementation. Communication overhead of distributed memory setting limits the performance of hybrid algorithm as well.

2.6.2 Speedup factors of our improved parallel algorithm DPLAL

Our final parallel implementation of Louvain algorithm is DPLAL. This algorithm achieves a speedup factor up-to 12. We reduce the communication overhead in message passing setting to a great extent by introducing a load balancing scheme during graph partitioning. The improved speedup for DPLAL is presented in Figures 2.6a, 2.7a and 2.8a. For smaller networks [Figure 2.6a], the algorithm scales to a couple of hundred processors. For larger networks [Figures 2.8a, 2.7a], our algorithm scales to a larger number of processors. We are able to use around a thousand processors. It is understandable that for smaller networks, the communication overhead gradually offsets the advantage obtained from parallel computation. However, since we want to use a significant number of processors to work on larger networks, our algorithm in fact has this desirable property. Overall, DPLAL algorithm scales well with the increase in the number of processors and to large networks.

2.6.2.1 Runtime analysis: a breakdown of execution times

We present a breakdown of executions times in Fig. 2.3 showing the runtime analysis for network RoadNet-PA. We observe that communication time for gathering neighbor information and exchanging duality resolved community decreases with increasing number of processors. Communication time for both exchanging updated community and gathering updated communities increases up-to a certain number of processors and after decreasing, the time becomes almost constant. Among all these communications, time for gathering updated communities at the root processor takes maximum time and contributes to the high runtime.
Figure 2.3: Runtime analysis of RoadNet-PA graph with DPLAL algorithm for varied number of processors. We show a breakdown of execution times for different modules or functions in the algorithm. Time for gathering updated communities and total duration are plotted w.r.t the right y-axis.

2.6.2.2 Network size versus execution time.

For many large networks that we experiment on (including the ones in Fig. 2.7a and 2.8a), we find that those can scale to up to $\approx 800$ processors. We call this number as the optimum number of processors for those networks. This optimum number depends on network size. As our focus is on larger networks, to find out the relationship between runtime and network size, we keep the number of processor 800 fixed and run an experiment.

As shown in Fig. 2.4, the communication time for gathering neighbor info decreases with growing network size whereas both time for gathering updated communities and exchanging duality resolved community increase. Communication time for exchanging updated community increases up-to a certain point and then starts decreasing afterwards. For larger networks ($> 80K$), total runtime increases proportionately with growing network size. As smaller graphs do not scale to 800 processors, those do not follow the trend. However, it can be inferred that those networks will behave the same way for their optimum number of processors.
Figure 2.4: Increase in runtime of DPLAL algorithm with an increase in the sizes of the graphs keeping the number of processors fixed (800 processors).

2.6.2.3 METIS partitioning approaches

We also compare the METIS partitioning techniques, edge-cut and communication volume minimization, to find out the efficient approach for our algorithm.

Fig. 2.5 shows the runtime comparison between edge-cut and communication volume minimization techniques for some of the networks. We find that the communication volume minimization approach always takes similar or higher time than that of edge-cut partitioning. So, in our subsequent experimentation, we have used edge-cut partitioning approach.

2.6.3 Performance Analysis

We show the performance of our algorithm against other prominent algorithms in the literature. We compare the performance of DPLAL with two other distributed memory parallel implementations of Louvain method.
Figure 2.5: Comparison of METIS partitioning approaches (edge-cut versus communication volume minimization) for several networks. The edge-cut approach achieves better runtime efficiency for the above real-world networks.

Figure 2.6: Speedup factors of DPLAL and VITE algorithms for small networks. Speedup curve of VITE completely follows a decreasing trend, whereas DPLAL shows good speedup.
2.6.3.1 Comparison with Vite [1, 2]

We compare the performance of DPLAL with Vite. We do not have access to the system for experimentation where the performance of Vite has been evaluated. So, we cannot directly compare the numerical figures given in papers [1, 2] because the performance of MPI varies with the system architecture. Therefore, we re-run Vite in our own environment so that we can keep the performance comparison consistent. We have also considered the baseline of Vite only because we have not used any heuristics in our algorithm.

![Speedup curve for DPLAL](a)

![Speedup curve for Vite](b)

Figure 2.7: Speedup factors of DPLAL and VITE algorithms for large networks. Speedup curve of Vite completely follows a decreasing trend, whereas DPLAL shows good speedup.

![Speedup curve for DPLAL](a)

![Speedup curve for Vite](b)

Figure 2.8: Speedup factors of DPLAL and VITE algorithms for larger networks. Larger networks scale to a larger number of processors. In case of Vite, speedup increases up to 256 processors maximum and decreases afterwards. Speedup curve of DPLAL follows increasing trend.

From Figures 2.6 and 2.7, we find that graphs upto 1.1M does not show any speedup for Vite [Figures 2.7b and 2.6b], whereas for DPLAL we get around $6 \times$ to $12 \times$ speedup [Figures 2.7a and 2.6a]. For larger graphs
beyond this limit, Vite shows moderate speed up for upto 256 processors and later on shows a decreasing trend of speedup with increasing number of processors [Figure 2.8b]. In case of DPLAL, the speedup follows an increasing trend with increasing number of processors [Figure 2.8a].

In Figure 2.8, for network roadnet-CA, Vite does not show any speedup but DPLAL achieves around 5\times speedup for this network. For soc-LiveJournal1 network as well, DPLAL shows higher speedup compared to Vite. For com-Orkut network both DPLAL and Vite shows comparable speedup and the maximum speedup gained from Vite and DPLAL is around 4.

The multi-core architecture of the processor plays an important role to determine the performance of the parallel algorithms. Considering the limited resource available to us during the experiments, DPLAL shows better performance compared to Vite in case of the moderate sized networks. For larger networks the performance of DPLAL is similar or better compared to Vite.

### 2.6.3.2 Comparison with Distributed Louvain by Charith et al. [3]

For a network with 500,000 nodes, the distributed parallel Louvain algorithm given in Charith et al. [3], achieves a maximum speedup of 6 whereas with DPLAL for a network with 317,080 nodes we get a speedup of 42 using 800 processors. The largest network processed by them has 8M nodes and achieves a speedup of 4. Our largest network achieves a comparable speedup (4-fold speedup with 5M nodes). The work in [3] have not reported runtime results so we could not compare our runtime with theirs directly. Their work have reported scalability to only 16 processors whereas our algorithm is able to scale to almost a thousand of processors.

### 2.6.4 Code Profiling: Performance Gain of DPLAL over Our Load imbalanced Naive Distributed Louvain Algorithm

We present a comparative analysis of our distributed memory algorithms based on the MPI Communication patterns using MPI Profiling tool TAU.

Using graph partitioner METIS, we have improved the performance of DPLAL compared to our initial distributed Louvain algorithm. From Figure 2.9 we can see the improved runtime of the major functions of the algorithms. DPLAL achieves 1281\times, 176\times, 805\times, and 15\times gain over the naive distributed Louvain algorithm in terms of runtime for func-1, func-2, func-3 and func-4 respectively. For the total runtime the gain is 291\times.

We use MPI Profiling toolkit TAU to understand the MPI communication patterns for our distributed louvain algorithms. Figure 2.10 shows the percentage of time spent by various MPI functions with different
Figure 2.9: DPLAL gains significant performance gain over our Naive Distributed Louvain algorithm. Here the functions are denoted as follows.
func-1: gathering neighbour info
func-2: exchanging updated community
func-3: exchanging duality resolved community
func-4: gathering updated communities

Figure 2.10: Percentage of Time spent in the MPI Functions in DPLAL
While the number of processors is fewer, time spent in $\text{MPI\_Send()}$ is $3.03 \times$ lesser compared to more processors being used. Similar trend is observed for $\text{MPI\_Recv()}$. Time spent in fewer number of processors is $1.35 \times$ lesser compared to the higher number of processors. In case of $\text{MPI\_Probe()}$, the number of processors do not influence the time and the time taken for $\text{MPI\_Probe()}$ is almost similar. Percentage of time spent in $\text{MPI\_Barrier()}$ is $1.23 \times$ higher while using more processors. We have several break points in the code where we need to use $\text{MPI\_Barrier()}$ to maintain the synchronization in the code flow. The major breakpoints are in between the primary functions, mentioned in Figure 2.9.
For `MPI_Send()` and `MPI_Recv()` functions 65% and 69% of the processors respectively, take less than average time to complete their task when the load is balanced given in Figures 2.11 and 2.12. From Figures 2.11 and 2.12 we also observe that load imbalanced `MPI_Send()`, `MPI_Recv()`, `MPI_Barrier()` and `MPI_Probe()` functions are $430.1 \times$, $392.6 \times$, $34.5 \times$ and $919.7 \times$ slower than the balanced approach respectively.

Figure 2.13: MPI Communication pattern in our initial Naive Distributed Louvain Algorithm

Figure 2.14: MPI Communication pattern in DPLAL

From Figures 2.13 and 2.14 we find that during MPI communications, only few processors have higher number of communications [Figures 2.13a and 2.14a] and larger message size compared to the larger threshold in the load imbalanced environment.

2.6.5 Discussion

In this work we have provided an efficient parallel Louvain algorithm in distributed memory setting. We show the performance of our algorithm for different real-world networks. We have identified some of the bottlenecks of the MPI-based parallel algorithm using profiling tool and overcome those in our solution DPLAL. We also provide a detailed design of our distributed memory algorithm which can be helpful for newcomers in High Performance Computing (HPC) to design distributed parallel algorithms for similar problems in many other domains. We compare the performance of our algorithm with other prominent
algorithms in literature. Scalability of the parallel algorithm and MPI performance is largely dependent on
the system architecture. We have a relatively small architecture available to us compared to the system
architectures in many other works. We have run Vite, a prominent algorithm from literature on the same
architecture in our system, LONI QB2 for same networks to measure the efficiency of our algorithm. LONI
QB2 is $11 \times$ and $4.7 \times$ smaller in terms of computing nodes compared to NERSC Edison and NERSC Cori
respectively on which Vite is tested. Our algorithm, DPLAL achieves better(in most cases) or comparable
scalability compared to Vite. This indicates that DPLAL will show better scalability if it is run on a larger
system architecture.

2.7 Conclusion

Our parallel algorithm for Louvain method demonstrates good speedup on several types of real-world graphs.
Shared memory system has limited number of physical cores and might not be able to process very large
networks. A large network often requires distributed processing where each computing node stores and
works with a part of the entire network. As we plan to work with networks with billions of nodes and
edges, we work towards the improvement of the scalability of our algorithms by reducing the communication
overhead. We have identified the problems for different implementations of our parallel algorithms and come
up with an optimized implementation DPLAL. We have shown a detailed design analysis of our MPI based
parallel algorithm that is useful for beginners in HPC to design parallel algorithms for similar problems
in other domains as well. With our improved algorithm DPLAL, community detection in DBLP network
achieves a 12-fold speedup. Our largest network, soc-LiveJournal1 has 4-fold speedup for same number
of processors. With increasing network size, number of processors also increase. The optimum number of
processor largely depends on the network size. We compare the performance of our algorithm with some
other well-known parallel algorithms in the literature. DPLAL achieves better speedup in most cases in a
relatively small system compared to the architectures of other supercomputers. We want to experiment with
other load balancing schemes to find more efficient load balancing scheme to make DPLAL more scalable
and optimized in future.
Chapter 3

Understanding Evolving Community Structures in Temporal Graphs using Dynamic Community Discovery

Temporal graphs are convenient mathematical abstraction for many practical complex systems. We present an analysis to see how communities evolve over times based on several graph metrics according to their temporal definitions. We also compare 6 different dynamic community detection (DCD) algorithms in terms of the quality of the output community and runtime. We find that the permanence-based method, DyPerm depending on a vertex-centric metric, works similarly efficient as DynaMo which is based on modularity. Permanence allows for local optimization and such local computation helps in parallelizing the technique without incurring a significant communication overhead. We present a scalable algorithm, ParDyPer for community discovery in dynamic graphs and to the best of our knowledge, this is the first parallelization of permanence-based method for detecting communities in large dynamic graphs. Our shared-memory parallel algorithm achieves 4 – 18 fold speed-up for several real-world graphs from different domains as well as synthetic graphs.

3.1 Introduction

Community evolution is important to understand the structures of the complex networks. This can be helpful to detect any drastic change in the interaction patterns. For certain networks it can also be useful
to predict the future trends of the network at certain time periods. These motivate us to perform this study to understand the evolving community natures throughout the snapshots in temporal networks.

Appropriate methods to detect temporal communities is an open problem. Many existing works are mostly done on synthetic networks. Batch edge addition-deletion on static graph at a distinct rate, does not represent the actual real-world scenario. Few studies have worked with large-scale networks. In this study we aim at observing different dynamic community detection methods. We want to compare the output community derived from each of the methods and their performance on same networks.

Based on our observations, we choose permanence, a vertex-based metric to design a parallel community detection algorithm, ParDyPer in dynamic networks. Permanence follows local optimization rather than global optimization of the full network. No arbitrary tie-breaking (inaccurate or insignificant communities with high score) scenario comes up in assigning communities like other functions (i.e. modularity). It is computationally expensive to directly employ a well-studied static algorithm repeatedly on the network snapshots of the evolving networks.

The contribution of our work is three fold.

- We identify the change of communities in different time phases computing several graph metrics based on their temporal definition for networks with ground-truth communities.

- We compare 6 different dynamic community detection (DCD) methods. We show how the output community differs in different methods depending on the quality of the community based on some of the graph metrics.

- Finally, We present a shared memory parallel algorithm for community detection in dynamic graphs using permanence, a vertex-based graph metric for different real-world and synthetic networks from different domains.

The rest of this paper is organized as follows. We describe the related works in Section 3.3. In Section 3.2 we describe in brief the background of temporal networks and the definition of several graph metrics used for community evolution in Section 3.6. In Section 3.4, we describe our methods to implement a shared memory parallel algorithm for temporal community detection using permanence. We also describe different steps we have performed to understand the evolving community structures per snapshot. Our datasets and experimentation environment are described in Section 3.5. A detailed analysis of the results is presented in Section 3.6. Finally, Section 3.7 summarizes the findings and concludes the paper.
3.2 Preliminaries

Notations, Definitions, and computational model used are described in this section.

3.2.1 Notation

For dynamic graphs we denote the input network as a collection of multiple network snapshots over time. There is a change of network in these multiple snapshots for different time-frames. We consider multiple edges appear and disappear in different time-frames. So, the full dynamic network can be represented as $G = G_0 \cup G_1 \cup G_2 \ldots \cup G_t \cup G_{t+1}$ A network snapshot at time $t$ is denoted by $G_t(V_t, E_t)$, where $V_t$ and $E_t$ are the sets of vertices and edges at time $t$, respectively. Vertices are labeled as $V^0, V^1, \ldots, V^{n-1}$. We use the words node and vertex interchangeably as well as links and edges.

3.2.2 Temporal Community Measures

In this section, we have described in brief the graph metrics [132] we use to evaluate the community structure. Some metrics only use the internal connectivity among the community members and shown in Table 3.1. The external connectivity metrics and their definitions are listed in Table 3.2. Graph metrics those consider both internal and external connectivity have been shown in Table 3.3.

Table 3.1: Temporal Community Measures Depending on Internal Connectivity Used for Community Evolution

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intra Community Edges</td>
<td>$E_{\text{intra}}$</td>
<td>Edges between the members of C</td>
</tr>
<tr>
<td>Internal Density</td>
<td>$\frac{2E_{\text{intra}}}{n_c(n_c-1)}$</td>
<td>Internal edge density of the nodes in community C</td>
</tr>
<tr>
<td>Average Degree</td>
<td>$\frac{2E_{\text{intra}}}{n_c}$</td>
<td>Average internal degree of the members of C</td>
</tr>
<tr>
<td>Fraction over median degree (FOMD)</td>
<td>$\frac{</td>
<td>{u \in C</td>
</tr>
</tbody>
</table>

3.2.3 Community Detection with Permanence

Permanence is a local vertex-based metric used to extract the community structure of large networks based on permanence optimization. It is better in the sense that it can be computed locally and does not require global optimization such as other modularity. It outperforms well-known community detection methods in
Table 3.2: Temporal Community Measures Depending on External Connectivity Used for Community Evolution

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inter Community Edges</td>
<td>$E_{\text{inter}}$</td>
<td>Number of edges on the boundary of $C$</td>
</tr>
<tr>
<td>Expansion</td>
<td>$\frac{E_{\text{inter}}}{N_c}$</td>
<td>Number of edges per node that point outside the community</td>
</tr>
<tr>
<td>Cut Ratio</td>
<td>$\frac{E_{\text{inter}}}{N_c(N-N_c)}$</td>
<td>Fraction of existing edges (out of all possible edges) leaving the community</td>
</tr>
</tbody>
</table>

Table 3.3: Temporal Community Measures Depending on Both Internal and External Connectivity Used for Community Evolution

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized Cut</td>
<td>$\frac{2E_{\text{intra}}+E_{\text{inter}}}{2(E-E_{\text{intra}})+E_{\text{inter}}}$</td>
<td>Computes the cut cost as a fraction of the total edge connections to all the nodes in the graph</td>
</tr>
<tr>
<td>Conductance</td>
<td>$\frac{E_{\text{inter}}}{2E_{\text{intra}}+E_{\text{inter}}}$</td>
<td>Fraction of total edge volume that points outside the community</td>
</tr>
<tr>
<td>Clustering Coefficient</td>
<td>$CC(u) = \frac{n_i}{\left(\frac{1}{2}D(u)\right)}$</td>
<td>Measures the “clumpiness” of a graph, ratio of the existing edges and the total number of possible edges among the neighbors of a node. Where, $n_i$ : number of edges between the neighbors</td>
</tr>
<tr>
<td>Separability</td>
<td>$\frac{E_{\text{intra}}}{E_{\text{inter}}}$</td>
<td>Measures the ratio between the internal and the external number of edges on the boundary of $C$</td>
</tr>
</tbody>
</table>

**Permanence of vertex $u$:**

$E_{\text{max}}(u)=2$

$I(u) = 4, d(u) = 6$

$C_{\text{in}}(u) = \frac{3}{4c} = \frac{3}{6} = 0.5$

$Perm^C(u) = \left[\frac{4}{2} \times \frac{1}{6}\right] - \left[1 - \frac{3}{6}\right] = \frac{1}{6}$

$= -0.167$

**Permanence after (u-v) edge deletion:**

$E_{\text{max}}(u)=2$

$I'(u) = 3, d'(u) = 5$

$C'_{\text{in}}(u) = \frac{3}{3c} = \frac{3}{3} = 1$

$Perm'^C(u) = \left[\frac{3}{2} \times \frac{1}{5}\right] - \left[1 - 1\right] = \frac{3}{10}$

$= 0.3$

Figure 3.1: Calculation of Permanence using an example graph
Table 3.4: Symbols used for calculating Permanence in Equation 2.1

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Perm(v)$</td>
<td>Permanence of vertex $v$</td>
</tr>
<tr>
<td>$I(v)$</td>
<td>Internal neighbors of vertex $v$</td>
</tr>
<tr>
<td>$E_{\text{max}}(v)$</td>
<td>Maximum connections to a single external community</td>
</tr>
<tr>
<td>$C_{\text{in}}$</td>
<td>Internal clustering coefficient</td>
</tr>
<tr>
<td></td>
<td>$C_{\text{in}} = \frac{x}{\binom{I(v)}{2}}$ means total number of possible connections among the internal neighbors of $v$</td>
</tr>
<tr>
<td>$Perm(v) = 1$</td>
<td>$v$ strongly connected to assigned community</td>
</tr>
<tr>
<td>$Perm(v) = 0$</td>
<td>$v$ equally pulled by all neighbors (singleton community)</td>
</tr>
<tr>
<td>$Perm(v) = -1$</td>
<td>$v$ weakly connected to assigned community (wrong community)</td>
</tr>
</tbody>
</table>

Figure 3.2: Workflow for Community Evolution and Parallel Community Detection in Temporal Graphs for different networks

terms of runtime complexity as shown in [133]. Permanence, $Perm$ is calculated using Equation 3.1, where $-1 < Perm < 1$. The meanings are described in Table 3.4. Figure 3.1 demonstrates a simple example of calculating permanence.

$$Perm(v) = \left[ \frac{I(v)}{E_{\text{max}}(v)} \times \frac{1}{d(v)} \right] - [1 - C_{\text{in}}]$$  \hspace{1cm} (3.1)

3.2.4 Computational Model

We develop a share-memory based parallel algorithm using Python Joblib and Multiprocessing modules similar to C++ Open Multi-Processing (OpenMP).

3.3 Related Work

Community detection is a very well known problem in graph mining for a long time [134, 135, 114, 111, 126, 136]. There exists several works on temporal community detection using different methods [27, 28] in literature. We find very few works similar to ours that we perform in this study. Authors in [20] focus
on progressively evolving graphs only in their study. They experiment on very small synthetic graphs with higher number of snapshots. On average those synthetic networks have only 100 nodes and 1200 steps (snapshots). They have shown evaluation of smoothness of dynamic partitions (snapshots). They have compared the output community based on modularity [134], AMI, ARI. For AMI, ARI calculation what they consider as ground truth is not mentioned in the work. In our work we include both real-world as well as synthetic networks from different domains. In our analysis, we select networks from varied size range, starting with a few hundred nodes increasing upto millions of nodes and edges. We compare the community quality depending on 12 different graph metrics. We also include the runtime comparison of the 6 DCD methods and choose the best performing algorithm to present a scalable parallel algorithm for dynamic community detection, ParDyPer.

Another work [29] shows Community quality evaluation at each time slice based on a single graph metric conductance. The value of conductance is compared for each community across the time slices in order to measure how community structure changes over time. Other than this, no prior works analyzed the community structure quality over the snapshots. In our case we include 12 different graph metrics to measure how community evolution is taking place per snapshot. In [30] the authors have compared static community detection methods only for temporal social networks whereas we compare dynamic community detection methods for networks from multiple domains.

Overall, we have contributed to observe the community quality per snapshot based on 12 different graph metrics. We have used both real-world and synthetic networks from multiple domains. We have shown the comparison of the output community structure for different dynamic community detection methods. We have identified the best performing method in terms of output quality and performance and design our parallel algorithm ParDyPer for dynamic community detection.

3.4 Methodology

In this section we describe different steps that we have performed in our study to observe evolution of ground truth communities in certain networks and the design of our parallel DCD algorithm ParDyPer.

3.4.1 Community Evolution for Ground Truth Communities

3.4.1.1 Pre-Processing Input Graph

For our experiments using dynamic networks, we use the edge-list format of the networks. The input consists of the tuple (source-node,destination-node,timestamp). We get a number of snapshots from the input graphs
based on the timestamp value and the duration of the network. Networks having connections from multiple years are divided into yearly snapshots and in some cases we divide the snapshots on monthly basis. For some networks having ground truth communities per snapshot, and for the synthetic networks, this step is not required.

### 3.4.1.2 Mapping Ground Truth Communities per Snapshot

In order to identify the evolution of ground truth communities throughout the snapshots, we map the communities with one another in all of the snapshots based on jaccard score. We consider choosing 10 communities from the last snapshot and back trace them in the previous snapshots. We select the 10 communities in the last snapshot based on the largest number of members in each community. We select the communities from the last snapshot because the communities are more stable in the last snapshot. Whereas the communities tend to deform more as those can be found in the initial snapshot. This is shown with example in Section 3.6.1. Mapping ensures that the evolution of the same community is portrayed throughout the snapshots.

### 3.4.1.3 Evaluating Community Structures using Graph Metrics per Snapshot

Next we compute the graph metrics values described in Section 3.2.2 for each of the communities. Based on these values we evaluate the community structures per snapshot to understand the evolution of the ground truth communities.

### 3.4.2 Comparing Output Communities and Performance for Different DCD Algorithms

In our study we use 6 different DCD algorithms to compare the community quality of each algorithm based on the graph metrics described in Section 3.2.2. We also compare the runtime performance for each of the algorithms. The DCD algorithms that we have used in this study are explained in brief below.

#### 3.4.2.1 No Smoothing Louvain (NoSL)

In this method at each snapshot static Louvain [134] community detection method is applied.

#### 3.4.2.2 Smoothed Louvain (SmoL)

This method is different from NoSL in the way that in subsequent snapshots it uses the community structure from the previous snapshot instead of a whole new community structure derived from the Louvain method.
3.4.2.3 DynaMo

This is an adaptive and incremental algorithm for updating the community structure of evolving networks [137]. It works for non-overlapping communities. It follows a 2 step method. In the first step, it initializes an intermediate community structure, depending on the incremental network changes and the previous network community structure. In step 2, it repeats the last two steps of Louvain method (i.e., Local Modularity Optimization and Network Compression) on the intermediate community structure until no modularity gain improvement is possible.

3.4.2.4 Estrangement Confinement (EsCon)

This method is based on estrangement metric [138], a measure of partition distance based on the inertia of inter-node relationships. When estrangement is constrained it provides meaningful temporal communities.

3.4.2.5 Transversal-Network (TraN)

In this method [136] communities at snapshot $t$ depends on the previous $t-1$ and later $t+1$ snapshots of the network. So, a single transversal network is developed by adding inter-snapshot coupling links. On this network an adapted version of modularity optimization algorithm is applied.

3.4.2.6 DyPerm

DyPerm [133] requires the initial network information and the community structure to detect communities in the subsequent snapshots based on the addition and removal of nodes or edges in the subsequent snapshots.

3.4.3 Overview of the Shared-Memory Algorithm: ParDyPer

In our shared memory implementation, we parallelize the algorithm computational task-wise in a straightforward approach, primarily loop parallelization. Whenever there is a need to iterate over the full network or even the neighbors of a node, considering the large network size, the work is done by multiple threads to minimize the workload and do the computation faster.
Figure 3.3: Edge Addition, Edge Deletion and Vertices for different networks in different time-slices

3.5 Experimental Setup

3.5.1 Environment

To run the experiments for the community evolution, we have used Desktop computer with the configurations of Intel Core i7-4770 CPU 3.40 GHz × 8 Processor, 16 GB of RAM, 1 TB hard disk, and Ubuntu 16.04 LTS.

For our shared memory implementation, we use Python Joblib and Multiprocessing modules to parallelize the tasks in a multi-threaded environment. We have used Louisiana Optical Network Infrastructure (LONI) to perform the experiments. QB2 [129], a 1.5 Petaflop peak performance cluster containing 504 compute nodes with over 10,000 Intel Xeon processing cores of 2.8 GHz has been used for the experiments performed. We can use a maximum number of 20 threads as QB2 is limited by 20 cores per node. Our approach is similar to the C++ OpenMP implementation.
Figure 3.4: Community Size (a-d) and Intra Community Edge (e-f) for different networks in different time-slices. For Intra Community Edge, Syn-1 and Syn-3 networks show constant insignificant values (0/1) over all snapshots.
3.5.2 Dataset

We use both real-world and synthetic networks for our experiments. The real-world networks are depicted in Table 3.5. For synthetic networks we use the dynamic LFR benchmark model [146] with parameters Number of Vertices \((N)\), Mixing Coefficient \((\mu)\) and Number of Snapshots \((s)\). The parameters for the synthetic networks are given in Table 3.6. The changes of vertices and edges per snapshot is shown in Figure 3.3.

3.6 Result

3.6.1 Understanding the Evolution of Ground Truth Communities based on different Graph Metrics

Community mapping in each of the snapshots is crucial to understand the community evolution over snapshots. We observe that communities in the last snapshot are more stable compared to the first snapshot. So we have backtraced the stable communities from the last snapshot to the first snapshots to track their evolution. We have shown this for the ground truth communities for Primary School network given in Figure 3.5.

![Figure 3.5: Community mapping using jaccard coefficient for the ground truth communities of Primary School network](image)

If we map communities with respect to the first snapshot, all communities have very low value of jaccard coefficient in the following snapshots observed in Figure 3.5a. Instead, if we map communities with respect to the last snapshot, the communities show better values for the jaccard coefficient in the previous snapshots.
<table>
<thead>
<tr>
<th>Domain</th>
<th>Network</th>
<th>Vertices</th>
<th>Edges</th>
<th>Snapshots</th>
<th>Time Interval</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collaboration</td>
<td>Cumulative Co-authorship</td>
<td>708,497</td>
<td>1,166,376</td>
<td>17</td>
<td>Cumulative Years</td>
<td>Cumulatively Aggregated (year) undirected co-authorship network in DBLP repository from 1960 to 2009 (vertex:author, edge:a pair of authors are co-authors at least once)</td>
<td>[139]</td>
</tr>
<tr>
<td>Non-Cumulative Co-authorship</td>
<td>708,497</td>
<td>1,166,376</td>
<td>17</td>
<td>1 Year</td>
<td>Undirected co-authorship network in DBLP repository from 1960 to 2009 (vertex:author, edge:a pair of authors are co-authors at least once)</td>
<td>[140]</td>
<td></td>
</tr>
<tr>
<td>Social</td>
<td>College-Msg</td>
<td>1,899</td>
<td>59,835</td>
<td>7</td>
<td>193 days (month)</td>
<td>Messages on a Facebook-like platform at UC-Irvine (vertex:user, edge:private message between users at t timestamp) This network focuses on users’ activity in the forum rather than private messages exchanged among users (vertex:user, edge:users’ activity in the forum)</td>
<td>[141]</td>
</tr>
<tr>
<td></td>
<td>fb-forum</td>
<td>899</td>
<td>33,720</td>
<td>7</td>
<td>193 days (month)</td>
<td>Contacts between the children and teachers</td>
<td>[142]</td>
</tr>
<tr>
<td>Primary School</td>
<td>242</td>
<td>77,602</td>
<td>6</td>
<td>20 seconds</td>
<td>Contacts between the children and teachers</td>
<td>[143, 144, 145]</td>
<td></td>
</tr>
<tr>
<td>Citation</td>
<td>cit-HepTh</td>
<td>22,768</td>
<td>352,807</td>
<td>7</td>
<td>10 years (year)</td>
<td>Arxiv HEP-TH (high energy physics theory) citation graph is from arXiv and covers all the citations from April 1993 to 2003. (vertex: paper, edge:a pair of papers) Edges from u to v indicate that a paper u cited another paper v</td>
<td>[142]</td>
</tr>
</tbody>
</table>
Figure 3.6: Internal Density (a-c) and Average Degree (d-f) for different networks in different time-slices. For Cumulative CoAuthorship network the values of Internal Density are considered negligible being very small (0.01 – 0.15). Syn-3 Network follows the similar pattern as Syn-1 Network for the values of Average Degree.
Table 3.6: Synthetic Networks used in the Experiment

<table>
<thead>
<tr>
<th>Network</th>
<th>$N$</th>
<th>$\mu$</th>
<th>$s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syn-1</td>
<td>3500</td>
<td>0.2</td>
<td>20</td>
</tr>
<tr>
<td>Syn-2</td>
<td>1000</td>
<td>0.2</td>
<td>30</td>
</tr>
<tr>
<td>Syn-3</td>
<td>1000</td>
<td>0.2</td>
<td>10</td>
</tr>
</tbody>
</table>

There is a mix up of high and low jaccard score found in Figure 3.5b. So we mapped communities in reverse order. This makes it easier to follow the evolution.

Community Size

In Primary School network [Fig. 3.4a], the selected communities are mostly increasing in size per snapshot. Only communities with id 4, and (8,9) shrinks in size in time-slice 2 and 6 respectively. In Cumulative CoAuthorship network [Fig. 3.4b], the selected communities increase in size (mostly linearly, exponentially) per snapshot. In Syn-1 network [Fig. 3.4c], the selected communities follow a mixed trend. The increase or decrease is very minimal for most communities. In Syn-3 network [Fig. 3.4d], the selected communities also follow a mixed trend similar to Syn-1 network. The increase or decrease varies within the range 2 to 9 for each snapshot.

3.6.1.1 Intra community edge

In Primary School network [Fig. 3.4e], the values of the intra community edge of the community members for the selected communities follow a mixed trend. All of the communities have a decrease in snapshot 2. A few decreases can be observed in snapshot 4 (community id 5,9). In snapshot 6, 50% of the communities have a decreased value and the rest remain constant (30%) or increase (20%). In Cumulative Co-Authorship network [Fig. 3.4f], the intra community edges for the selected communities increase mostly linearly, power (community id 0), and logarithmic (community id 1). The increase is more significant in the last two snapshots.

3.6.1.2 Internal Density

In Primary School network [Fig. 3.6a], the values of the internal edge density of the community members for the selected communities mostly follow a zig-zagged pattern. Half of the communities have values above average (0.5 to 1) indicating good quality of the communities throughout the snapshots.

In Syn-1 network [Fig. 3.6b], the selected communities follow mostly a decreasing trend of internal edge density of the community members over time, with a few spikes in some of the snapshots. In Syn-3 network
[Fig. 3.6b], the selected communities follow a similar pattern to Syn-1 network. Only communities with ids 0, 6, and 9 show a spike in snapshots 7, 4, and 8 respectively.

![Graphs of FOMD for different networks](image)

Figure 3.7: FOMD for different networks in different time-slices

### 3.6.1.3 Average Degree

In Primary School network [Fig. 3.6d], the average internal degree for the selected communities are mostly following the similar pattern as the intra community edge metric. It is correlated with intra community edge given in Fig. 3.4e. As many intra-connected edges are cut down, average degree reflects the decreasing nature and vice-versa. For Cumulative Co-authorship network[Fig. 3.6e], the change is very trivial and remains within the range $0.5 - 2.5$.

In Syn-1 network [Fig. 3.6f], most communities follow a decreasing trend of average internal degree of the community members over time, with a few spikes in some of the snapshots. The change is very minimal within the range of 1.3 to 2.4.
3.6.1.4 FOMD

In Primary School network [Fig. 3.7a], the values of FOMD for the selected communities follow almost similar pattern as the Intra Community Edge metric seen in Figure 3.4e. For the Cumulative Co-Authorship network [Fig. 3.7b], most of the communities share the same values and follows a decreasing trend over the snapshots. In both Syn-1 and Syn-3 networks, the values of the FOMD metric vary within a very small range, close to 0.

![Intra Community Edge for Primary School Network](image-a)

![Inter Community Edges for Cumulative Co-authorship Network](image-b)

![Inter Community Edges for Syn-1 Network](image-c)

![Inter Community Edges for Syn-3 Network](image-d)

Figure 3.8: Inter Community Edges for different networks in different time-slices

3.6.1.5 Inter Community Edge

In Primary School network [Fig. 3.8a], the inter community edges for the selected communities have a sharp increase and (decrease) in snapshots 2, 5 and (3, 6) respectively. In Cumulative Co-Authorship network [Fig. 3.8b], the inter community edges for the selected communities follow a mixed pattern including both
increasing and decreasing nature throughout the snapshots. In both Syn-1 [Fig. 3.8c] and Syn-3 [Fig. 3.8d] networks, the selected communities follow a mixed trend of both increasing and decreasing pattern but very small changes in values.

### 3.6.1.6 Expansion

In Primary School network [Fig. 3.11a], the values indicate positive correlation with cut ratio and normalized cut. The graph also follows similar pattern as given in Fig. 3.11c and Fig. 3.11e. In Cumulative CoAuthorship network [Fig. 3.11b], the value of expansion maintains a positive relation with cut ratio [Fig. 3.11d] and normalized cut [Fig. 3.11f].
3.6.1.7 Cut Ratio

In Primary School network [Fig. 3.11c], the values indicate positive correlation with expansion and normalized cut. The graph follows similar pattern as given in Fig. 3.11a and Fig. 3.11e. In Cumulative CoAuthorship network [Fig. 3.11d], the value of cut ratio maintains a positive relation with expansion [Fig. 3.11b] and normalized cut [Fig. 3.11f].

3.6.1.8 Normalized Cut

For both Primary School [Fig. 3.11e] and Cumulative CoAuthorship networks [Fig. 3.11f], the values and graph pattern is almost same as conductance given in Fig. 3.9a and Fig. 3.9b.
Figure 3.11: Expansion, Cut Ratio and Normalized Cut for different networks in different time-slices. Both Syn-1 and Syn-3 networks has almost constant and very small values for all of these metrics and considered negligible
Figure 3.12: Separability (a-b) and Permanence (c-f) for different networks in different time-slices. Intra Community Edge being 0 for most communities in Syn-1 and Syn-3 networks, the value of separability also becomes 0 for those communities.
3.6.1.9 Conductance

In Primary School network [Fig. 3.9a], the values indicate positive correlation with expansion, cut ratio and normalized cut. The graph also follows similar pattern as given in Fig. 3.11a, Fig. 3.11c and Fig. 3.11e. In Cumulative CoAuthorship network [Fig. 3.9b], the value of conductance maintains a positive relation with expansion [Fig. 3.11b], cut ratio [Fig. 3.11d] and normalized cut [Fig. 3.11f]. In both Syn-1 [Fig. 3.9c] and Syn-3 networks [Fig. 3.9d], the value of conductance is very low within the range 0 – 0.01, very close to 0. This small values indicate very strong connectivity among the member of the communities. It also shows the inverse relation with permanence given in Fig. 3.12e and Fig. 3.12f.

3.6.1.10 Clustering Coefficient

In Primary School network [Fig. 3.10a], the selected communities mostly follow the opposite pattern of expansion [Fig. 3.11a]/cut ratio [Fig. 3.11c]/normalized cut [Fig. 3.11e]/conductance [Fig. 3.9a] metrics as expected by the definition. We notice a few exceptions for communities with id 4,5. In Cumulative CoAuthorship network [Fig. 3.10b], the value of clustering coefficient follows an inverse relation with conductance [Fig. 3.9b]. In both Syn-1 [Fig. 3.10c] and Syn-3 networks [Fig. 3.10d], the value of clustering coefficient follows a decreasing trend throughout the snapshots.

3.6.1.11 Separability

In Primary School network [Fig. 3.12a], the positive correlation with permanence [Fig. 3.12c] and inverse correlation with expansion [Fig. 3.11a], cut ratio [Fig. 3.11c], normalized cut [Fig. 3.11e] and conductance [Fig. 3.9a] is maintained well. In Cumulative CoAuthorship network [Fig. 3.12b], positive correlation with permanence [Fig. 3.12d] is maintained except a few exception (comm-id 6).

3.6.1.12 Permanence

In Primary School network [Fig. 3.12c], the selected communities mostly follow the opposite pattern of expansion [Fig. 3.11a]/cut ratio [Fig. 3.11c]/normalized cut [Fig. 3.11e] metrics as expected by the definition. In Cumulative CoAuthorship network [Fig. 3.12d], the selected communities follow a mixed trend. If we follow some particular communities, the relation with the other graph metrics becomes prominent. For communities with ids 2,4 and 6 we observe that these have a sharp increase, decrease and decrease at snapshots 6, 5 and 4 respectively. We can observe from the values of conductance [Fig. 3.9a] that these same communities follow just the opposite pattern. In both Syn-1 [Fig. 3.12e] and Syn-3 networks [Fig. 3.12f], the
value of permanence is very high within the range $0.97 - 1$. This indicates very strong connectivity among the member of the communities.

After analyzing all 12 graph metrics to understand the community evolution for the ground truth communities, we observe that the metrics using both internal and external connections portray a better concept of community qualities. So, normalized cut, conductance and permanence takes the major role in providing more valuable insights about the community structure. In some cases, clustering coefficient is also helpful but shows some exceptions.

### 3.6.2 Comparison of Different Dynamic Community Detection Methods

<table>
<thead>
<tr>
<th>Network</th>
<th>DyPerm (sec)</th>
<th>NoSL (sec)</th>
<th>SmoL (sec)</th>
<th>DynaMo (sec)</th>
<th>EstCon (hrs)</th>
<th>TraN (hrs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary School</td>
<td>0.61</td>
<td>6.50</td>
<td>9.30</td>
<td>0.55</td>
<td>10.01</td>
<td>1.09</td>
</tr>
<tr>
<td>CollegeMsg</td>
<td>10.10</td>
<td>28.40</td>
<td>100.51</td>
<td>8.21</td>
<td>22.29</td>
<td>12.83</td>
</tr>
<tr>
<td>fb-forum</td>
<td>88.15</td>
<td>297.10</td>
<td>151.92</td>
<td>92.73</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>citHepTh</td>
<td>16896.09</td>
<td>18751.00</td>
<td>20900.47</td>
<td>18551.91</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cumulative CoAuthorship</td>
<td>3765.90</td>
<td>2971.60</td>
<td>4011.66</td>
<td>4197.10</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Non-cumulative CoAuthorship</td>
<td>3525.50</td>
<td>8193.70</td>
<td>6957.10</td>
<td>4212.97</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>syn-1</td>
<td>677.26</td>
<td>1889.54</td>
<td>1511.64</td>
<td>507.94</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>syn-2</td>
<td>2258.98</td>
<td>5076.01</td>
<td>3549.30</td>
<td>2713.04</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>syn-3</td>
<td>218.47</td>
<td>507.72</td>
<td>392.81</td>
<td>220.24</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

We have compared different dynamic community detection methods to show how the community structures change over time based on different methods. We have explained the change in community structure with the values of different graph metrics those help us to understand the quality of the community structure. We initially start with 6 dynamic community detection methods for comparison of the community structures throughout the snapshots. But two of the methods (EstCon, TraN) show increasingly long run times compared to the rest of the methods shown in Table 3.7. So, we decide to eliminate those two for further comparisons. From Figures 3.13, 3.15, 3.16 we observe that the community quality is similar as reflected in the similar values of the graph metrics. To keep it simple, we have shown the results for Conductance, Normalized Cut, Clustering Coefficient and Permanence because these metrics take into consideration both
Figure 3.13: Comparison of different DCD methods for a particular community in Primary School Network

to internal and external connectivity.

3.6.3 Parallel Dynamic Community Detection: ParDyPer

Figure 3.17 represents the speed-up for our shared-memory parallel algorithm ParDyPer. We get around 4× to 18× speed-up for different types of networks.
In this work we focus on investigating how community structure changes over time in different real-world temporal networks. We analyze the evolution of ground truth communities over the snapshots based on different graph metrics depending on the internal, external and both connectivity among the community members. We run our experiments on several real-world and synthetic networks from different domains. We also show a comparison between 6 different dynamic community detection methods. We verify the quality of the community structure derived from different methods w.r.t ground truth. We observe that for certain networks the quality of the community is almost similar using different methods. Our experiments show that runtime makes a major difference in choosing the algorithms for the detection of temporal communities. We
Figure 3.15: Comparison of different DCD methods for a particular community in College Message Network

find that DynaMo and DyPerm are comparable to each other in terms of runtime. Choosing “permanence” metric over “modularity” is prioritized to emphasis local optimization rather than global one. We choose to implement a parallel dynamic community detection based on DyPerm because of the advantages it provides being a vertex-centric metric. We implement a shared-memory parallel algorithm ParDyPer using multi-
Figure 3.16: Comparison of different DCD methods for a particular community in Facebook Forum Network threading and loop-parallelization. Our result shows 4 – 18 fold speed-up on different networks.
Figure 3.17: Speed-up for shared-memory parallel algorithm ParDyPer for different networks
Chapter 4

DyGMPCD: A Distributed Parallel Community Detection Algorithm for Large-scale Dynamic Graphs

Temporal (Dynamic) graph represents several real-world complex systems and community detection in dynamic graphs has several important applications. We detect communities based on a vertex-centric metric, permanence, that detects communities by local optimization rather than global optimization of the full network. To the best of our knowledge, there is no other distributed-memory parallel methods in current literature for detecting communities in dynamic networks. We present a novel parallel algorithmic framework, Dynamic Graph MPI-based Parallel Community Detection (DyGMPCD) for detecting communities in large-scale dynamic networks. We also include 3 heuristics to our baseline algorithm. Applying these heuristics improve the performance of our algorithm significantly preserving the solution quality. We experiment on several real-world large-scale networks from different domains. DyGMPCD achieves up to $30\times$ speedup for the largest network. Our algorithm also shows $30\times$ performance gain over a state-of-the-art parallel algorithm of the literature.

4.1 Introduction

Graphs are a very powerful representation of different real-world complex systems. In the traditional network mining, often the static topological properties of the graphs are used to solve different problems of graph
theory. But time plays an important role to accurately represent the complex networks like social networks (Facebook, Twitter), brain networks, transportation networks and many more which change over time. Appropriate methods to detect temporal communities is an open problem. We use permanence, a vertex-based metric to detect communities in dynamic networks. Permanence follows local optimization rather than global optimization of the full network. No arbitrary tie-breaking (inaccurate or insignificant communities with high score) scenario arises in assigning communities like other functions (i.e. modularity). It is computationally expensive to directly employ a well-studied static algorithm repeatedly on the network snapshots of the evolving networks. Existing works are mostly done on synthetic networks and few works focus on large-scale networks. To the best of our knowledge, there is no distributed-memory parallel algorithm using Message Passing Interface (MPI) to detect communities in dynamic graphs. We have worked towards developing an MPI-based parallel algorithm for community detection in dynamic graphs using permanence which faces the challenges of distributed-memory parallelization. We mitigate those challenges using different heuristics based on the structural property of the input graph.

Our Contributions: The main contributions of this paper are as follows:

- Presenting DyGMPCD, a distributed-memory parallel community detection algorithm for dynamic networks
- Design and evaluation of multiple heuristics that improve the performance of the baseline parallel algorithm
- Performance evaluation and assessment of the correctness of our solutions on several large-scale real-world networks (234 millions edges) from 6 different domains (social, blog, communication, collaboration, citation, biological)
- Comparison of DyGMPCD with existing parallel community detection algorithm for dynamic graphs shows better performance

4.2 Related Works

Community detection problem has a long history as one of the fundamental problems of graph mining. Several algorithms are used for community detection and Louvain method [134] is a novel one based on the modularity metric for static networks [147, 114, 111, 126]. Regarding community detection in dynamic networks, a few approaches have used the modularity metric using different variants of Louvain algorithm [136, 137]. Constraining estrangement metric [138], a measure of partition distance based on the inertia of
inter-node relationships, meaningful temporal communities can be found. DyPerm [133] uses permanence metric as measure and requires the initial network information and the community structure to detect communities in the subsequent snapshots. Authors in [148] use a find-and-merge type of community detection algorithm that can efficiently handle the streaming updates.

Some recent works on dynamic graphs focus on providing a framework [31, 32, 33, 34, 35, 36] for storing and managing the networks with high performance and efficiency to support different graph algorithms on GPUs and other various HPC platforms. Some of the parallel algorithms applied on dynamic networks [149] include breadth first search (BFS) traversal [37], finding dense subgraphs [38], shortest path [39], k-cores [40] and others. However, very few works have been done on the parallel community detection algorithm for dynamic networks. Halappanavar et. al [41] have presented a technique to detect communities on dynamic graphs and future work includes implementing the method in their shared-memory based parallel framework Grappolo. The implementation and experimental analysis of their mentioned technique is not available yet.

A shared-memory based multi-threaded community detection method for streaming graph is presented in [42]. The authors have implemented an incremental re-agglomeration algorithm using STINGER framework and Open Multi-Processing (OpenMP) that considers batch of changes to detect communities. The main limitation of the algorithm is that it cannot detect when a community is split into separate components. However, the algorithm shows 4× to 3700× speedup over static re-computation for all updates. But the authors have not provided the parallel speedup of the algorithm with increased number of threads.

The only other parallel algorithm for detecting communities in dynamic networks is a Spark-based implementation that shows 2× parallel speedup with 12 cores [43]. Their algorithm identifies incremental vertices and defines community membership depending on maximizing Parallel Weighted Community Clustering (PWCC) metric of entire network. They have worked with mostly the synthetic networks with at most 2 million vertices and 10 million edges. The only real-world network they use have 200K vertices and 500K edges. For the synthetic networks, the degree distribution and the community size is very small which mostly does not represent the real-world dynamic networks like large-scale social networks those are highly skewed.

To the best of our knowledge, we have implemented the first MPI-based distributed parallel community detection algorithm for dynamic networks. We have done experiments on several real-world large-scale networks from different domains. We cannot compare our work with [43] as the source code is not publicly available. They have shown around 2× parallel speedup whereas for our largest network we get 30× speedup. We also compare our work with the existing OpenMP-based parallel algorithm [42] and we get almost 30× performance gain compared to theirs.
4.3 Preliminaries

Notation, problem definition, and the computational model used in our solution are described in this section.

4.3.1 Notation

For dynamic graphs, we denote the input network as a collection of multiple network snapshots over time. There is a change of network in these multiple snapshots for different time-frames. We consider that multiple edges appear and disappear in different time-frames. So, the full dynamic network can be represented as $G = G_0 \cup G_1 \cup G_2 \ldots \cup G_t \cup G_{t+1}$. A network snapshot at time $t$ is denoted by $G_t(V_t, E_t)$, where $V_t$ and $E_t$ are the sets of vertices and edges at time $t$ respectively. Vertices are labeled as $V^0, V^1, \ldots, V^{n-1}$. We use the words node and vertex interchangeably as well as links and edges. $P$ is the number of processors used in the computation, denoted by $P_0, P_1, \ldots, P_N$ where $0, 1, 2, \ldots, N - 1$ refers to the rank of a processor. Terms frequently used throughout the paper, are enlisted in Table 4.1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G(V, E)$</td>
<td>Graph with $V =$ set of vertices and $E =$ set of edges</td>
</tr>
<tr>
<td>$G_t$</td>
<td>Graph network at time $t$</td>
</tr>
<tr>
<td>$C_x^t$</td>
<td>Community structure of network at time $t$ where $x$ is community-id</td>
</tr>
<tr>
<td>$C(v)$</td>
<td>Community of vertex, $v$</td>
</tr>
<tr>
<td>$d(v)$</td>
<td>Degree of vertex, $v$</td>
</tr>
<tr>
<td>$d_{max}$</td>
<td>Maximum degree</td>
</tr>
<tr>
<td>$\overline{d}$</td>
<td>Average degree</td>
</tr>
<tr>
<td>$N$</td>
<td>Total number of processors (World size)</td>
</tr>
</tbody>
</table>

4.3.2 Problem Description

Let $G_t(V_t, E_t)$ denotes the graph at time step $t$. Our goal is to compute the communities $C_k$ at time step $k$, based on the community structure $C_{k-1}$ at time step $k - 1$ and the changed edge set $\Delta E_k$.

4.3.3 Community Detection with Permanence

Permanence is a local vertex-based metric used to extract the community structure of large networks based on permanence optimization. It is better in the sense that it can be computed locally and does not require global optimization such as modularity. It outperforms well-known community detection methods in terms of runtime complexity as shown in [133]. Permanence, $Perm$ is calculated using Equation 4.1, where $-1 <$
Permanence of vertex $u$:
\[ E_{\text{max}}(u) = 2 \]
\[ I(u) = 4, d(u) = 6 \]
\[ C_{\text{in}}(u) = \frac{5}{2} \times \frac{5}{6} = 0.8333 \]
\[ Perm^C(u) = \left[ \frac{4}{2} \times \frac{1}{6} \right] - \left[ 1 - \frac{5}{6} \right] = \frac{1}{6} \]
\[ = 0.167 \]

Permanence after (u-v) edge deletion:
\[ E_{\text{max}}(u) = 2 \]
\[ I'(u) = 3, d'(u) = 5 \]
\[ C_{\text{in}}'(u) = \frac{5}{3} \times \frac{5}{3} = 1.667 \]
\[ Perm'^C(u) = \left[ \frac{3}{2} \times \frac{1}{5} \right] - \left[ 1 - \frac{5}{3} \right] = \frac{29}{30} \]
\[ = 0.967 \]

Figure 4.1: Calculation of Permanence using an example graph

$Perm < 1$. The meanings are described in Table 4.2. Figure 4.1 demonstrates a simple example of calculating permanence.

\[ Perm(v) = \left[ \frac{I(v)}{E_{\text{max}}(v)} \times \frac{1}{d(v)} \right] - [1 - C_{\text{in}}] \quad (4.1) \]

Table 4.2: Symbols used for calculating Permanence in Equation 4.1

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Perm(v)$</td>
<td>Permanence of vertex $v$</td>
</tr>
<tr>
<td>$I(v)$</td>
<td>Internal neighbors of vertex $v$</td>
</tr>
<tr>
<td>$E_{\text{max}}(v)$</td>
<td>Maximum connections to a single external community</td>
</tr>
<tr>
<td>$C_{\text{in}}(v)$</td>
<td>Internal clustering coefficient of vertex $v$</td>
</tr>
<tr>
<td>$C_{\text{in}}(v) = \frac{x}{C(I(v),2)}$</td>
<td>$C(I(v),2)$ means total number of possible connections among the internal neighbors of $v$</td>
</tr>
<tr>
<td>$Perm(v) = 1$</td>
<td>$v$ strongly connected to assigned community</td>
</tr>
<tr>
<td>$Perm(v) = 0$</td>
<td>$v$ equally pulled by all neighbors (singleton community)</td>
</tr>
<tr>
<td>$Perm(v) = -1$</td>
<td>$v$ weakly connected to assigned community (wrong community)</td>
</tr>
</tbody>
</table>

4.3.4 Computational Model

We develop the parallel algorithm for Message Passing Interface (MPI) based distributed-memory parallel systems, where each processor has its own local memory. The processors do not have any shared memory, one processor cannot directly access the local memory of another processor, and the processors communicate via exchanging messages using MPI.
Algorithm 10: DyGMPCD: Our MPI-based Parallel Dynamic Community Detection Algorithm

Data: Initial Graph $G_t(V_t,E_t)$, Current Graph $G_{t+1}(V_{t+1},E_{t+1})$, Initial Community $C_t$
Result: Community at next time-stamp $C_{t+1}$

1. edges _added $\leftarrow E_{t+1} - E_t$
2. edges _deleted $\leftarrow E_t - E_{t+1}$
3. Each processor $P_i$ does computation for a set of vertices $V^i$ (depending on partitioning strategy)
4. for Each mini-batch of edges do
   5. $E_{add} \leftarrow$ part of edges _added
   6. $E_{del} \leftarrow$ part of edges _deleted
   7. MPI_Scatter(root, $E_{add}$)
   8. MPI_Scatter(root, $E_{del}$)
   9. for Each processor $P_i$ (executing in parallel) do
      10. Edge Addition()
      11. MPI_Barrier()
      12. MPI_AllGather(comm_list)
      13. Edge Deletion()
      14. MPI_Barrier()
      15. MPI_AllGather(comm_list)
      16. MPI_Finalize()
   17. end
18. end
19. return $C_{t+1}$

4.4 Methodology

In this section we have described our algorithm DyGMPCD, the implementation challenges, the complexity analysis of DyGMPCD and the heuristics we use to improve the performance of DyGMPCD.

4.4.1 Overview of our Distributed-Memory Algorithm

Our aim is to compute the communities of the full network in a distributed manner. To serve the purpose, we distribute the full network among the processors in such a way that each processor can do its computation by having minimum communication to one other. After addition of all of the edges of the snapshot, the processors have to communicate with the root processor to update the network. Again, after deletion of all of the edges of the snapshot, all processors communicate with the root processor with the updated community list. Lastly, the root processor finalize the output community list.

We use an incremental approach to detect communities on subsequent snapshots based on the previous snapshot's community structure and the dynamic updates (addition and deletion of edges and vertices) those take place on the current snapshot.
Algorithm 11: Edge Addition

```plaintext
1 for Each edge e in edges_added, do
2     e = (u, v) /* Intra-Community */
3     if C(u) = C(v) then
4         G_t.add_edge(u, v)
5         Update_Neighbors()
6     end /* Inter-Community */
7     else
8         if C(u), C(v) ∈ P_i then
9             (perm, comm_list) ← calculate_permance()
10        end
11        else if C(u) ∈ P_i then
12            n ← u
13            P_j ← C(v)
14        end
15        else if C(v) ∈ P_i then
16            n ← v
17            P_j ← C(u)
18        end
19        (perm, comm_list) ← calculate_permance(n)
20        MPI_Send(perm, P_j)
21        MPI_Recv(perm, P_j)
22        if perm > perm2 then
23            MPI_Send(comm_list, P_j)
24            Update Neighbors()
25        end
26        else
27            MPI_Recv(comm_list, P_j)
28        end
29     end
30 end
```

4.4.1.1 Graph Partitioning with METIS

We have used the well-known graph-partitioner METIS [150, 151] to partition our input graph to distribute it among the processors. Initially we intend to use both edge-cut and communication volume minimization approaches. Since the design of our algorithm largely depends on the edge-cut minimization among the processors, we present the results of our experiments with such approach.

4.4.1.2 Community Detection

Algorithm 10 represents the pseudo-code of our distributed-memory parallel algorithm for community detection on dynamic networks. We process the total edge additions and deletions in mini-batches. Small
batches of edges are communicated to all processors by the root processor. Algorithm 11 and 13 describes
the addition and deletion of edges in the dynamic network respectively and the corresponding changes of
the community structure throughout the updates.

**Algorithm 12: Update Neighbors**

```plaintext
1  \( P_u \) \( \leftarrow \) A list of processors where neighbors of u belong to
2  \( P_v \) \( \leftarrow \) A list of processors where neighbors of v belong to
3  for \( \text{times} = 1 \) \( \text{to} N \) do
4      if \( P_j \) in \( P_u \) and \( P_v \) then
5          MPI_Send(1, \( P_j \))
6      else
7          MPI_Send(0, \( P_j \))
8      end
9  MPI_Recv (x, \( P_j \))
10  \( \text{degree}(u|v) + = x \)
```

4.4.1.2.1 *Edge Addition* There are two cases for adding edges in a dynamic graph. These are described
below.

- **Intra-Community Edges**

  Addition of intra-edges is straight-forward. The edge \((u, v)\) is added locally in the network. The
  internal neighbors of \(u\) & \(v\) are updated about this addition through MPI communication as shown in
  Algorithm 12.

- **Inter-Community Edges**

  Addition of inter-edges require much communication among the processors as illustrated in Algorithm
  11 to calculate the value of permanence as the neighbors of a vertex are scattered among multiple
  processors. If both the communities, \(C(u), C(v)\) of vertex \(u\) & \(v\) belong to the current processor \(P_i\),
  then we have to check whether \(u\) moves to new community depending on the permanence value. \(C_{new}(u)\)
  is then advertised to \(u\)’s internal neighbors in a BFS manner. If the neighbors of \(u\) are in different
  processors, we do a blocking communication here. Although we try to do a group communication with
  multiple processors for batch processing of multiple edges together, the difference in the permanence
  value lead to wrong community assignment. Therefore, we process each single edge instead of multiple
  edges together. For inter-processor neighbors of \(u\), the order is changed as all of those neighbors are
  processed after the intra-processor neighbors are done.
Algorithm 13: Edge Deletion

```
for Each edge e in edges_deleted, do
    e = (u,v)
    /* Intra-Community */
    if C(u) = C(v) then
        if degree(u) = 1 & degree(v) = 1 then
            G_t.remove_edge(u,v)
            C(u) ← C(u) − u − v
            comm_list ← comm_list ∪ u ∪ v
            if C(u) ∉ P_j then
                P_j ← C(u)
                MPI_Send(comm_list, P_j)
        end
    else if degree(u|v) = 1 then
        G_t.remove_edge(u,v)
        C(u) | C(v) ← C(u)| C(v) − v
        comm_list ← comm_list ∪ u|v
        Update Neighbors()
        if C(u)|C(v) ∉ P_j then
            P_j ← C(u)|C(v)
            MPI_Send(comm_list, P_j)
    end
    else
        non_unit_degree_edge_delete()
    end
    /* Inter-Community */
    else
        G_t.remove_edge(u,v)
    end
end
```

4.4.1.2.2 Edge Deletion  Edge deletion seems computationally less complicated compared to adding edges in dynamic graphs in terms of permanence calculation. There are two cases with multiple sub-cases to delete edges as shown in Algorithm 13.

- **Intra-Community Edges**

  There are mainly two sub-cases for intra-community edge deletions.

    - **Unit Degree Vertices** u & v are removed from the current community and u, v form their own singleton community. If the community of the vertices (u, v) or u or v belong to different processors, we communicate the updates to those processors.

    - **Non-Unit Degree Vertices** It is similar to the Inter-Community Edge Addition. The compu-
tations and the communications follow the same structure as Algorithm 11. The pseudo-code for this non-unit degree vertices is given in Algorithm 14.

- *Intra-Community Edges*

No computation of permanence is required in this step. Only communicating the update of deletion is needed if multiple processors are involved.

---

**Algorithm 14: non\_unit\_degree\_edge\_delete()**

```plaintext
1 if u, v ∈ P_i then
2   (perm, comm_list) ← calculate_permance()
3 end
4 else
5   if u ∈ P_i then
6     n ← u
7     P_j ← v
8 end
9 else if v ∈ P_i then
10    n ← v
11    P_j ← u
12 end
13 (perm, comm_list) ← calculate_permance(n)
14 MPI_Send(perm, P_j)
15 MPI_Recv(perm2, P_j)
16 if perm > perm2 then
17   MPI_Send(comm_list, P_j)
18   Update Neighbors()
19 end
20 else
21   MPI_Recv(comm_list, P_j)
22 end
```

---

### 4.4.2 Implementation

#### 4.4.2.1 Challenges

During the initial implementation of our algorithm, we have been using the Adjacency List (AL) format to store our input network using the C++ Standard Template Library (STL) containers vector and map. We have avoided the Compressed Sparse Row (CSR) format because it takes $O(d_{max})$ in the worst case to remove an edge from the graph whereas the time complexity is $O(1)$ for AL representation. However, the performance of our DyGMPCD algorithm gets limited due to the dynamic edge update operations. We have tried with both sorted and unsorted neighbor list for the vertices. The linear search over the unsorted
neighbors and the binary search over the sorted neighbors have similar time as given in Figure 4.2. So the major challenge we face is to select a better data structure to minimize the search time while finding edges for calculation and adding/deleting edges for the dynamic updates.

![Figure 4.2: Time (%) spent in dynamic edge updates from the overall computation time for the sorted and unsorted list using AL and PMA. The time is normalized w.r.t PMA.](image)

Keeping the communications synchronized is another challenge of this algorithm to maintain the correctness. For this reason we have to use the MPI_Barrier after the addition phase and deletion phase to get a consistent result.

### 4.4.2.2 Selection of Efficient Data Structure

Storing the dynamic networks is a separate and complex problem itself. Based on the dynamic updates, different types of data structures are considered efficient for storing such temporal information. We consider selecting such structure that helps us improving the performance of the algorithm by minimizing the update time of the network. After exploring different dynamic graph storing framework [33, 36, 152] to find such efficient storage, we find that using Packed-Memory Array (PMA) is a very good alternative [33] to use for skewed graphs like social networks. We also want to observe how PMA works for networks from other domains as well. So, we have used a sorted array for small-degree vertices and PMA for large-degree vertices to store our networks. From Figure 4.2 we see that the update time is reduced to almost half after using PMA.
4.4.3 Complexity Analysis

We analyze the run-time complexity of our distributed-memory MPI-based Algorithm 10. Suppose, $E_a$, $E_d$ be the number of edges to be added & deleted for the full network respectively and $p$ be the number of partitions. For Algorithm 11, in the calculation of permanence [Line 9], the computational complexity in worst case scenario would be: $O(d^2 (\log d)^2)$ and for the best case: $O(d (\log d)^2)$ where, $d$ denotes degree. The communication complexity would be $O(\alpha \overrightarrow{E})$ where $\alpha$ is a multiplying factor and $\overrightarrow{E}$ denotes number of cut-edges between partitions. Line 19 also has the same computational and communication complexity. So, combining these, the average complexity for edge-addition is: $O \left( \frac{E_a}{p} \left[ d^2 (\log d)^2 + \alpha \overrightarrow{E} \right] \right)$. In case of edge deletion, in Algorithm 13, for line [4-22], there is only communication complexity that is the same as edge addition. Again, for Line 24, the complexity is the combination of computational and communication complexity similarly as given in Algorithm 11, Line 19. So, the total run-time complexity of the algorithm becomes:

$$O \left( \frac{E_a}{p} \left[ d^2 (\log d)^2 + \alpha \overrightarrow{E} \right] + \frac{E_d}{p} \left[ d^2 (\log d)^2 + \alpha \overrightarrow{E} \right] \right)$$

So, the time complexity of DyGMPCD will be higher for both dense and skewed networks.

4.4.4 Heuristics for Performance Optimization

4.4.4.0.1 Heu-1: Pseudo Edge Deletion with active/inactive labelled edges

We keep one additional parameter to identify edges those exist in current snapshot. We tag the edge labels as “Active” and “Inactive” to reduce the exact edge deletion operations. When an edge is being deleted, we change the edge label from “Active” to “Inactive” in that particular snapshot. So, we only need to do the search operation instead of removing the element from the data structure. Again, in most networks, a deleted edge may reappear in the subsequent snapshot. Therefore, the inactive edge is changed back to active with the search and update operation only instead of the insertion operation.

4.4.4.0.2 Heu-2: Compressing Graph Size by Pruning Small Degree Nodes

In the community detection problem, 1-degree or 2-degree vertices do not contribute to the output communities as they mostly act as separate entity in the network. As they are completely isolated in the network, we can reduce the network size to a great extent if we can eliminate such vertices from our calculation. In a sense, we are compressing the network size by keeping them out of the scope of computation. This heuristic can be mostly applicable to networks from the social/blog/communication network domains where some vertices are hub vertices and the network is skewed. So the scope of evolving for most small vertices gets
4.4.4.0.3  **Heu-3: Computational Efficiency by Excluding Smaller Communities**

We take into consideration the community distribution of the network at each of the snapshots. While we find singleton communities contributing to a higher percentage, we keep those communities separate in the “Inactive Community List”. For the computation of permanence (selecting communities), the computation cost cuts down, as the number of active members in the community list is minimized.

### 4.5 Experimental Setup

#### 4.5.1 Environment

We have used both NERSC and Louisiana Optical Network Infrastructure (LONI) supercomputers to perform the experiments. Cori is a Cray XC40 system comprised of 2,388 Intel Xeon “Haswell” processor nodes, and 9,688 Intel Xeon Phi “Knight’s Landing” (KNL) nodes. QB3 [153], a 857 Teraflop peak performance cluster containing 202 compute nodes connected by 100 Gbps Infiniband fabric with 9,696 CPU cores. 24-core Intel Cascade Lake (Intel® Xeon® Platinum 8260 Processor) CPUs have been used for the experiments.

#### 4.5.2 Dataset

We use both real-world dynamic and synthetic dynamic updates to large-scale static networks for our experiments. The real-world dynamic networks are depicted in Table 4.4. We also experiment with some large-scale static networks by doing dynamic updates (insertion and deletion) of edges in batches. Details of these networks are listed in Table 4.3. We include the synthetic dynamic updates because available actual real-world dynamic networks are comparatively smaller in size.

In Figure 4.3, we have shown the network statistics, i.e. number of vertices, edges, communities and dynamic updates per snapshot for the dynamic real-world networks. We choose one network from each of the domains. From Figure 4.4, we observe that except the ‘Youtube’ network, rest of the networks grows in size in subsequent snapshots. The percentage change of addition of edges is higher than that of deletion. Figures 4.3 and 4.4 help the readers to learn about the network and how the networks are evolving over time at a glance.
4.6 Experimental Evaluation and Performance

We have discussed the performance of our algorithm in terms of speedup as well as the output community quality in this section. We also describe the optimized performance of our algorithm by applying different heuristics. We also show the comparison of our algorithm with other state-of-the-art algorithms of literature.
Table 4.4: Real-world Dynamic Networks used in the Experiment

<table>
<thead>
<tr>
<th>Domain</th>
<th>Network</th>
<th>Vertices</th>
<th>Edges</th>
<th>(\bar{d})</th>
<th>(d_{max})</th>
<th>Snapshots</th>
<th>Time Interval</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Social</td>
<td>primary school</td>
<td>242</td>
<td>77,602</td>
<td>1,000</td>
<td>2,600</td>
<td>6</td>
<td>20 secs</td>
<td>[141]</td>
</tr>
<tr>
<td></td>
<td>College-Msg</td>
<td>1,899</td>
<td>20,296</td>
<td>3</td>
<td>1,000</td>
<td>7</td>
<td>month</td>
<td>[141]</td>
</tr>
<tr>
<td></td>
<td>Youtube</td>
<td>3,200,000</td>
<td>12,200,000</td>
<td>7</td>
<td>129,800</td>
<td>8</td>
<td>year</td>
<td>[142]</td>
</tr>
<tr>
<td>Blog / Forum</td>
<td>sx-stackoverflow</td>
<td>2,601,977</td>
<td>63,497,050</td>
<td>5</td>
<td>100,000</td>
<td>9</td>
<td>year</td>
<td>[141]</td>
</tr>
<tr>
<td></td>
<td>sx-superuser</td>
<td>194,085</td>
<td>1,443,339</td>
<td>7</td>
<td>11,200</td>
<td>9</td>
<td>year</td>
<td>[141]</td>
</tr>
<tr>
<td></td>
<td>sx-mathoverflow</td>
<td>24,818</td>
<td>506,550</td>
<td>3</td>
<td>7,800</td>
<td>8</td>
<td>year</td>
<td>[141]</td>
</tr>
<tr>
<td></td>
<td>fb-forum</td>
<td>899</td>
<td>33,720</td>
<td>74</td>
<td>1,800</td>
<td>6</td>
<td>month</td>
<td>[142]</td>
</tr>
<tr>
<td>Communication</td>
<td>wiki-talk</td>
<td>1,140,149</td>
<td>7,833,140</td>
<td>3</td>
<td>22,300</td>
<td>8</td>
<td>year</td>
<td>[141]</td>
</tr>
<tr>
<td></td>
<td>email-Eu-core</td>
<td>986</td>
<td>332,334</td>
<td>4</td>
<td>1,000</td>
<td>8</td>
<td>year</td>
<td>[141]</td>
</tr>
<tr>
<td></td>
<td>email-enron</td>
<td>87,000</td>
<td>1,148,073</td>
<td>26</td>
<td>38,700</td>
<td>3</td>
<td>year</td>
<td>[141]</td>
</tr>
<tr>
<td>Collaboration</td>
<td>non-cumulative co-Authorship</td>
<td>708,497</td>
<td>1,166,376</td>
<td>2</td>
<td>11,200</td>
<td>17</td>
<td>year</td>
<td>[141]</td>
</tr>
<tr>
<td></td>
<td>cumulative co-Authorship</td>
<td>708,497</td>
<td>1,166,376</td>
<td>6</td>
<td>4,300</td>
<td>17</td>
<td>year</td>
<td>[141]</td>
</tr>
<tr>
<td>Citation</td>
<td>HepTh</td>
<td>22,900</td>
<td>2,700,000</td>
<td>233</td>
<td>12,000</td>
<td>7</td>
<td>year</td>
<td>[141]</td>
</tr>
<tr>
<td></td>
<td>HepPh</td>
<td>28,100</td>
<td>4,600,000</td>
<td>327</td>
<td>11,100</td>
<td>10</td>
<td>year</td>
<td>[141]</td>
</tr>
</tbody>
</table>

4.6.1 Scalability

We measure the performance of our algorithm by speedup (ratio of the sequential and parallel execution time). Figure 4.5 represents the speedup for our distributed-memory parallel algorithm for networks from different domains. We get \(12−25\times\) speedup for different types of networks with base version of the algorithm using maximum 1024 processors. We get the maximum \(25\times\) speed-up for ‘Orkut’, the largest social network in our dataset.
Figure 4.3: Network statistics for different real-world dynamic networks in different time-slices

From Figure 4.6 we observe that our algorithm gains more scalability for larger networks. For relatively small networks (having edges in between ten thousands and one million), the average speedup is $5 \times$. Networks having edges in the range of 1 million to 100 million, shows around $10^{-15} \times$ speedup. For relatively larger networks, upto 1 billion edges, our distributed algorithm shows $20 - 25 \times$ speedup.

4.6.2 Performance Gain using Different Heuristics

We achieve significant performance gain using all of the heuristics depicted in Figures 4.7, 4.9 and 4.11.

4.6.2.1 Heu-1: Pseudo Edge Deletion with active/inactive labelled edges

Our first heuristic, Heu-1, is applicable to all domains of networks. We have shown the performance gain of our parallel algorithm for the major networks from each domain in Figure 4.7. We get a maximum of 20% performance gain in speedup for the ‘Patents’ dataset. For our largest dataset ‘Orkut’, we have around
Figure 4.4: Percentage change of edge addition and deletion for different real-world dynamic networks in different time-slices

16.5% performance gain, and the speedup increases up to 28.

4.6.2.2 Heu-2: Compressing Graph Size by Pruning Small Degree Nodes

We apply our second heuristic, Heu-2 to only those networks that have smaller degrees in all of the snapshots. In Figure 4.8, we have shown the degree distribution of the ‘sx-stackoverflow’ network. We can observe that for snapshots 3 up to 8, the number of vertices having degree 0/1/2 comprises around 50% – 60% of the total vertices. In snapshots 1 and 2, 30% and 35% of the total vertices have these smaller degrees. In the last snapshot 9, the percentage of smaller degrees is much higher and 70% of the vertices have these small-sized neighbors. The other networks, we choose to apply the Heu-2 heuristic, shows similar kind of degree distribution. Figure 4.9 shows the performance gain using our Heu-2 heuristic. We get around $26 \times$ speedup for the ‘sx-stackoverflow’ dataset. For ‘Orkut’ network, the performance gain is around 20% and we have the maximum speedup 30 amongst all of our datasets.
Figure 4.5: Speedup for our distributed parallel algorithm for multiple networks from different domains. For networks having edges more than a million, shows scalability for up to 1024 processors (maximum no. of processors used for experiments). For networks with less than a million edges, shows scalability for 64 processors only and do not scale more after this point.

Figure 4.6: Scalability of our parallel algorithm for increasing network size (number of edges) observed for 1024 processors.
Figure 4.7: Performance gain of DyGMPCD using heuristic, Heu-1 observed for 1024 processors. Heuristic, Heu-1 is based on active/inactive edge labels to decide on the presence or absence of edges at each time slices. Green color denotes the improved speedup gained by running the Heu-1 heuristic based algorithm, whereas the base speedup shown with the orange color is obtained by running our main algorithm without using any heuristics.

Figure 4.8: ‘sx-stackoverflow’ is a representative network with huge number of smaller degrees (degree: 0-3) in most of the snapshots. The percentage of degree distribution according to number of degrees is given by the stacked bars in the left figure (a). In the right figure (b), the average degree of the network per snapshot is shown by the blue bars.
Figure 4.9: Performance gain of our parallel algorithm using heuristic, Heu-2 and combining both Heu-1 and Heu-2 heuristics observed for 1024 processors. The left figure (a) shows the change in speedup. The right figure (b) shows the performance gain in percentage.

Figure 4.10: ‘wiki-talk’ is a representative network with huge number of singleton communities in most of the snapshots. The percentage of community distribution according to community size is given by the stacked bars in the left figure (a). In the right figure (b), the average number of communities of the network per snapshot is shown by the blue bars.

4.6.2.3 Heu-3: Computational Efficiency by Excluding Smaller Communities

We apply our third heuristic, Heu-3 to only those networks which have singleton communities in all of the snapshots. In Figure 4.10, we have shown the community distribution of the ‘wikitalk’ network. We can observe that for all the snapshots around 95% of the total communities are singleton. Figure 4.11 shows the performance gain using the H3 heuristic. We get around $14 \times$ speedup for the ‘wikitalk’ dataset. If we apply both Heu-2 and Heu-3 heuristics together, the improvement is around 36% and the speedup becomes 15.
Figure 4.11: Performance gain of our parallel algorithm using heuristic Heu-3 and the combination of all Heu-1, Heu-2 and Heu-3 heuristics observed for 1024 processors. The left figure (a) shows the change in speedup. The right figure (b) shows the performance increase in percentage.

Again, if we apply Heu-1, Heu-2 and Heu-3 heuristics altogether, the improvement is around 48% and the overall speedup becomes around 16.

4.6.3 Qualitative Analysis of the Community Structure

Table 4.5: Deviation in permanence value after applying the heuristics, Heu-2 and both (Heu-1+Heu-2) together

<table>
<thead>
<tr>
<th>Permanence % Change</th>
<th>wiki-talk</th>
<th>stackoverflow</th>
<th>Orkut</th>
<th>wiki-talk</th>
<th>stackoverflow</th>
<th>Orkut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>0.6976</td>
<td>0.4300</td>
<td>0.4676</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Heu-2</td>
<td>0.6972</td>
<td>0.4296</td>
<td>0.4676</td>
<td>0.0624</td>
<td>0.1001</td>
<td>0.0094</td>
</tr>
<tr>
<td>Heu-1+Heu-2</td>
<td>0.6967</td>
<td>0.4291</td>
<td>0.4669</td>
<td>0.1283</td>
<td>0.2001</td>
<td>0.1549</td>
</tr>
</tbody>
</table>

Table 4.6: Deviation in permanence value after applying the heuristics, Heu-1, Heu-2 and Heu-3 on wiki-talk network

<table>
<thead>
<tr>
<th>Permanence</th>
<th>Change</th>
<th>%Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>0.69759</td>
<td>0</td>
</tr>
<tr>
<td>Heu-3</td>
<td>0.69737</td>
<td>0.00022</td>
</tr>
<tr>
<td>Heu-2+Heu-3</td>
<td>0.69712</td>
<td>0.00047</td>
</tr>
<tr>
<td>Heu-1+Heu-2+Heu-3</td>
<td>0.69659</td>
<td>0.00100</td>
</tr>
</tbody>
</table>

We compare the output communities of both sequential and parallel algorithms comparing the accuracy
Figure 4.12: Comparison of the output community quality in between the sequential and parallel methods based on permanence. Applying heuristic, Heu-1 keeps the permanence value unchanged as well.

of the permanence metric based on which we derive the output communities. From Figure 4.12 we can observe that the overall quality of the output communities remain same in the parallel algorithm as the sequential one. In Table 4.5 we present the deviation in permanence value after applying heusitic, Heu-2 and both Heu-1 and Heu-2 heuristics together. We find that the deviation is very little as 0.2% and can be considered negligible. In Table 4.6 we compare the permanence value by applying all three heuristics for the ‘wiki-talk’ network. We can observe that the deviation is as little as 0.00100 only after applying three of the heuristics together.

4.6.4 Comparison with the State-of-the-art Algorithms

In this section we have shown the comparison of our parallel algorithm DyGMPCD with the only 2 existing parallel algorithms in current literature.

4.6.4.1 STINGER shared-memory re-agglomeration algorithm

We have compared our distributed-memory based parallel algorithm, with the only shared-memory parallel algorithm implemented using the STINGER framework. The authors have presented efficiency of the parallel algorithm in terms of throughput. So we also present the same measure to compare our algorithm with
Regarding the throughput calculation, the runtime includes edge updates and community detection computational time. We have used the same network to compare the performance of both algorithms. As the OpenMP implementation is exploiting thread efficiency, we can use maximum 32 threads, the number of physical cores available to the system. We report our maximum processors’ performance for the comparison. From Figure 4.13, we can observe that our algorithm can handle more computations using the same number of batch size. The best throughput value observed from STINGER Parallel re-agglomeration algorithm is 0.2 million updates per second (10K batch size). Using 1024 processors, our distributed parallel algorithm can process 6.45 million updates per second for 100K batch size. So our algorithm is able to process the dynamic updates 30× faster compared to the STINGER parallel re-agglomeration algorithm.

![Figure 4.13: Comparison of Parallel Performance based on Throughput [Updates per second by threads (OpenMP)/processors(MPI)] for coPapersDBLP network. Legends present different batch sizes. Left Figure (a) shows the throughput for STINGER parallel re-agglomeration algorithm. Right Figure (b) shows the throughput for our distributed parallel algorithm DyGMPCD.](image)

4.6.4.2 Parallel Weighted Community Clustering (PWCC) Maximizing Spark-based Algorithm

We cannot directly compare our work with [43] as the source code is not publicly available. They have shown around 2× parallel speedup whereas for our largest network we get 30-fold speedup.

4.7 Conclusion

We have developed a scalable and optimized distributed MPI-based parallel algorithm, DyGMPCD for community detection on large-scale dynamic graphs. Since the runtime of our algorithm is a factor of cut-edges
across the partitions, a good partitioning strategy minimizing the cut-edges can make our algorithm more scalable. For several real-world networks from 6 different domains (social, blog, communication, collaboration, citation, biological), our algorithm performs well with maximum $30\times$ speedup for the largest network in our dataset. We apply 3 different heuristics that improve the performance of our algorithm by at most $48\%$ in terms of speedup. For accuracy, DyGMPCD shows deviation as little as $0.009 - 0.2\%$ without compromising the quality of the solution. Our parallel algorithm also achieves $30\times$ performance gain compared to the only shared-memory parallel algorithm implemented using STINGER framework for community detection on dynamic networks.
Chapter 5

Scalable Neural Networks

5.1 Community Detection using Semi-supervised Learning with Graph Convolutional Network on GPUs

Graph Convolutional Network (GCN) has drawn considerable research attention in recent times. Many different problems from diverse domains can be solved efficiently using GCN. Community detection in graphs is a computationally challenging graph analytic problem. The presence of only a limited amount of labelled data (known communities) motivates us for using a learning approach to community discovery. However, detecting communities in large graphs using semi-supervised learning with GCN is still an open problem due to the scalability and accuracy issues. In this study, we present a scalable method for detecting communities based on GCN via semi-supervised node classification. We optimize the hyper-parameters for our semi-supervised model for detecting communities using PyTorch with CUDA on GPU environment. We apply Mini-batch Gradient Descent for larger datasets to resolve the memory issue. We demonstrate an experimental evaluation on different real-world networks from diverse domains. Our model achieves up to 86.9\% accuracy and 0.85 F1 Score on these practical datasets. We also show that using identity matrix as features, based on the graph connectivity, performs better with higher accuracy than that of vertex-based graph features. We accelerate the model performance 4 times with the use of GPUs over CPUs.

5.1.1 Introduction

Community detection in graphs (or graph clustering) is an important graph analysis kernel with many real-world applications in diverse socio-technical domains including sociology, biology, infrastructure, web,
and epidemiology [154, 114, 155]. For example, clustering helps studying the effect of rumor or epidemic spreading in social or population network. Functional units in protein-interaction networks can also be found by community detection [156]. In general, community detection kernel identifies large-scale map of a network where individual communities act like meta-nodes or functional units. Communities reveals significant insights on the organization, function, or structural characterization of a system represented by graph [154].

Nowadays, we experience a huge growth of data generating from data-driven scientific and technical disciplines, e.g., online social media, biological sciences, business systems, and the Internet [157]. Such large datasets (including graph/network data) necessitate scalable methods for efficient and effective sense-making [141, 158].

Due to the advancement of data technology and computing resources, machine learning, particularly deep learning, has become a popular and exciting area of research for a growing number of applications [82, 54, 5, 109]. Graph Convolutional Network (GCN) combines both graph algorithms and deep learning and provides a wide avenue of practical applicability. GCN has demonstrated great promise in mining graph data (citation, social, and protein-protein interaction networks) [54], predicting protein interface [44], disease classification [45], molecules discovery in chemistry, text classification [46], identifying traffic congestion, image classification [49, 50], etc.

Using semi-supervised learning we can achieve the same or comparable result with less computational cost compared to a traditional community detection method. Community detection has several applications where this computational cost is high because of large datasets. Estimating unknown features of users/entities in social networks is a common application of community detection. If we think of Facebook network, it has over 2.6 billion active users. Traditional community detection algorithms need to analyze the full network to detect the communities and is computationally expensive. In case of semi-supervised learning, the main advantage is that if we have community information of a part of the network, we can predict the rest communities depending on those community labels for such large networks.

In this work, we use graph convolutional network to detect communities in large graphs using semi-supervised node classification. Our key contributions are as follows.

- **Model:** We present a scalable method for detecting communities via semi-supervised node classification. We apply Mini-batch Gradient Descent to solve the memory issues of GCN for larger and denser networks. We use Identity Matrix as Feature set and achieve better performance compared to node-based Graph Feature Set.
• **Data:** We experiment with a diverse set of real-world networks from different domains. Our model achieves around 86.9% accuracy and 0.85 F1 Score. For the popular datasets, our model performs comparably with the state of the art and even better in several cases.

• **Experiment:** We provide an extensive and thorough experimental evaluation of our model. We perform an optimization on the hyper parameters for better accuracy. We demonstrate a comparison of our method with related work empirically. We also accelerate the model performance 4 times using GPUs.

The rest of this paper is organized as follows. We describe the existing related works in Section 5.1.2. Preliminary introduction on Graph Convolutional Network is presented in Section 5.1.3. We describe our methods to generate features and train the model in Section 5.1.4. A detailed analysis of the results, e.g., description of our datasets, experimental setting, model performance, optimization of the hyper parameters, improvement of model accuracy, model speedup with parallelization, etc., are described in Section 5.1.5.

### 5.1.2 Related Works

There has been a rich literature of community detection in networks [158, 159, 160, 155, 154, 161, 108, 111, 114]. Most of these works are computationally expensive and do not scale well to large-sized networks. Kipf et al. [54] demonstrate some promising results based on GCN for graph clustering, albeit using a small dataset (Zachary’s karate club network). Both modularity-based clustering [162] and untrained 3-layer GCN Model [54] detect the same clusters for the said network. The authors [54] did not extend their work for large and more diverse datasets.

Another work by Chen et al. [51] used graph neural network (GNN) to detect communities with supervised learning. The paper restricted the largest community size to 800 nodes. They used networks having average community size below 30 in their experiments. Thus, an extensive experimentation with large-scale graphs with variety of domain and structural properties is still missing in literature.

GraphSAGE [5] is a framework designed to efficiently generate representations on evolving graphs with previously unseen data leveraging node attribute information. The mean aggregator used by the paper is nearly equivalent to the convolutional propagation rule used in the GCN framework [54]. In experimentation, they omit the largest communities from the networks. LAyer-Dependent ImportancE Sampling (LADIES) [163] is also a sampling based method. This algorithm selects their neighborhood nodes depending on the sampled nodes in the upper layer and constructs a bipartite subgraph. Then, the method computes the importance probability accordingly and samples a fixed number of nodes based on this probability. But the
authors limited their experimentation to a few small-sized dataset. Another framework, Neural Overlapping Community Detection (NOCD) is a graph neural network model for overlapping community detection [164]. They use the Bernoulli–Poisson (BP) model that allows for overlapping communities. One major difference between their model and GCN [54] is that they apply batch normalization after the first graph convolution layer. Another distinction is $L^2$ regularization being applied to all weight matrices.

Our work is unique from the above works based on the facts that we have used large-scale networks in our experiments and have not limited the number of communities as done in [51, 5]. We also compare our work with GraphSAGE [5] and our model shows better performance (as presented in Section 5.1.5.4). We have done a comprehensive experimentation taking networks from diverse domains whereas only small-sized citation networks were used in previous work [54]. Thus, our method is not prone to any bias that a particular domain or class of dataset might introduce. We have also used vertex-based metrics to generate Graph Feature Set and have made prediction on this feature set as well. In addition, we have made our model scalable to large graphs and compute systems using GPUs.

Cluster-GCN [52] is a SGD-based (Stochastic Gradient Descent) algorithm to design the minibatches based on efficient graph clustering algorithms. This method restricts the neighborhood search within the subgraph and achieves good memory and computational efficiency. GraphSAINT [53] is another graph sampling based inductive learning method that constructs minibatches by sampling the training graph, rather than the nodes or edges across GCN layers. This method ensures extracting appropriately connected subgraphs so that little information is lost when propagating within the subgraphs. In this method information of many subgraphs are combined together so that the training process overall learns good representation of the full graph. Our work differs from both in choosing the minibatches. In our work we do the sampling based on dense neighborhoods and keep track of the nodes in two consecutive layers. In our sample networks given in the experiments, the communities or labels vary within a large range, but these two methods experiment on networks with a limited number of labels.

5.1.3 Preliminaries

In this section, we present a brief introduction to Graph Convolutional Network (GCN). GCN is a neural network that operates on graphs. GCN utilizes the graph structure and collects node information from the neighborhoods in a convolutional manner. Given a graph $G = (V, E)$, a GCN takes as input

- an input feature matrix, $X$ of $N \times F$ dimension, where,
  
  $N =$ number of nodes in graph $G$ and
  
  $F =$ number of input features for each node, and

101
• an adjacency matrix, $A$ $(N \times N)$ of graph $G$.

There is a hidden layer $H^i = f(H^{i-1}, A)$ where $H^0 = X$ and $f$ is a propagation rule. Each layer $H^i$ corresponds to an $N \times F^i$ feature matrix. Features are aggregated to form the next layer’s features using the propagation rule $f$. The output label is denoted by $Y$.

GCN approaches fall into the following two categories:

■ Spatial-based: Feature information from local neighbors are aggregated.
  – Recurrent-based: Steady states of nodes are collected.
  – Composition-based: Higher orders of neighborhood information are obtained.

■ Spectral-based: Noise from graph signals are removed. An Eigen decomposition of the Laplacian Matrix of the graph is performed.

Here we will focus on Spectral GCNs only. For an $L$-layer GCN, the layer-wise propagation rule can be written as Eqn. 5.1.

$$Z^{(l+1)} = A'X^{(l)}W^{(l)}, X^{(l+1)} = \sigma(Z^{(l+1)}), \quad (5.1)$$

where, $X^{(0)} = X$ and $Z^{(L)} = Y$. The activation function $\sigma(.)$ is often the element-wise ReLU function. Weight Matrix $W^{(l)} \in \mathbb{R}^{F \times F}$ is the feature transformation matrix. $A'$ is the normalized and regularized adjacency matrix. For semi-supervised node classification, we have to minimize the error over all labeled examples. The error function is given in Eqn. 5.2. We use Cross-entropy Error as the loss function.

$$\mathcal{L} = \frac{1}{|Y|} \sum_{i \in Y} \text{loss}(Y_i, z_i^{(L)}) = -\frac{1}{|Y|} \sum_{i \in Y} \sum_{c=1}^{M} Y_{i,c} \ln p_{i,c} \quad (5.2)$$

Here, $M$ is number of node-labels/class. $Y_i$ is predicted probability observation. $p_{i,c}$ denotes $Y_i$ is of class $c$.

5.1.4 Methodology

In this section, we describe our machine learning model by presenting the methods for feature generation, training, testing and validation.

5.1.4.1 Computational Tools/Libraries

We use PyTorch [165] for deep learning models. Initially, we use Gephi [166] for feature generation from node statistics of the graph. Subsequently, we use GraphVis: Interactive Visual Graph Mining and Machine
Learning Tool [142] to export node-based features from the graphs. We also use Weka [167] for selecting attributes after generating node-based feature set. Pytorch 1.0.1 has been installed within Anaconda [168] 4.6.11 environment. Python scikit-learn [169, 170] module is used for performance evaluation of the model.

5.1.4.2 Model Classifier

We follow two steps to build our classifier. In the first step, we generate features from the datasets. We train the model using the generated features in the second step. Both of these steps are described in the subsequent sections.

Table 5.1: Node-based Feature Set Generated Using Gephi

<table>
<thead>
<tr>
<th>No.</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Degree</td>
<td>the number of edges connected to the node</td>
</tr>
<tr>
<td>2</td>
<td>Triangle</td>
<td>the number of times a node is included in forming a triangle</td>
</tr>
<tr>
<td>3</td>
<td>Clustering Coefficient (Watts-Strogatz)</td>
<td>a measure of how complete the neighborhood of a node is</td>
</tr>
<tr>
<td>4</td>
<td>Betweenness</td>
<td>measures how often a node appears on shortest paths between nodes in the network</td>
</tr>
<tr>
<td>5</td>
<td>Bridging Coefficient</td>
<td>the average probability of leaving the direct neighbor subgraph of a node</td>
</tr>
<tr>
<td>6</td>
<td>Bridging Centrality</td>
<td>a node centrality index based on information flow and topological locality in networks; product of the betweenness centrality and the bridging coefficient</td>
</tr>
<tr>
<td>7</td>
<td>Eccentricity</td>
<td>the maximum graph distance between a node and any other vertex of the graph</td>
</tr>
<tr>
<td>8</td>
<td>Eigen Centrality</td>
<td>a measure of node importance in a network based on a node’s connections</td>
</tr>
<tr>
<td>9</td>
<td>Pagerank</td>
<td>measures the importance of each node within the network</td>
</tr>
<tr>
<td>10</td>
<td>Authority</td>
<td>indicates the value of the node itself</td>
</tr>
<tr>
<td>11</td>
<td>Hub</td>
<td>estimates the value of the links outgoing from the node</td>
</tr>
</tbody>
</table>

5.1.4.2.1 Feature Generation We have used two types of feature sets. One feature set is Identity Matrix where the model is aware of the identity of each node by a unique one-hot vector. The identity matrix is of $N \times N$ dimension, where, $N$ is the number of nodes. A one-hot vector is a $1 \times N$ matrix (vector), being used to represent the encoding of each node in the graph. The vector consists of 0s in all cells with the exception of a single 1 in a cell used uniquely to identify each node. Using the identity matrix as the feature matrix results in highly local representations of each node, i.e., nodes that belong to the same
area of the graph are likely to be embedded closely together. In case of distant areas it is difficult for the network to share knowledge in an inductive fashion. In spite of this difficulty, results are comparable to other embeddings using more expensive unsupervised training procedure.

Another feature set is based on node statistics of the network. We use Gephi to generate this feature set. The attributes for this feature set is shown in Table 5.1. After generating the features we normalize the features using Weka to keep consistency among the values. While working with larger datasets, we cannot use Gephi to get the node statistics because of graph size limitations in Gephi. Gephi can support graphs upto 1 million nodes and 1 million edges. As we are experimenting with graphs larger than that size, to generate the node statistics we use another tool, *GraphVis*: Interactive Visual Graph Mining and Machine Learning Tool. Another reason to use GraphVis is that using the features generated by GraphVis show better accuracy compared to that of Gephi. For GraphVis, we use the features numbered 1, 2, 3, 4, 7, and 9 from Table 5.1, same as Gephi. Additional features generated by GraphVis is given in Table 5.2.

<table>
<thead>
<tr>
<th>No.</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>kcore-number</td>
<td>The core number of a node is the largest value k of a k-core containing that node.</td>
</tr>
<tr>
<td>b</td>
<td>4-clique</td>
<td>Number of 4-cliques formed by the vertex</td>
</tr>
<tr>
<td>c</td>
<td>4-chordal-cycle</td>
<td>Number of 4-chordal-cycle formed by the vertex</td>
</tr>
<tr>
<td>d</td>
<td>4-tailed-triangle</td>
<td>Number of 4-tailed-triangle formed by the vertex</td>
</tr>
</tbody>
</table>

5.1.4.2.2 Training Model At first, we apply the Louvain algorithm to our input graph to generate the community of each node for using as label to train the model. We use a diverse set of data for which ground-truth community is not available. Louvain algorithm is a well-established algorithm and widely used for community detection [154, 158, 159]. It is an efficient heuristic for community detection and used for large-scale datasets using the parallel variants of Louvain method [41, 171, 108]. So, we detect the communities from this also compares our graph datasets using Louvain algorithm and use as the ground-truth for each network. The community distribution found from the Louvain algorithm is used for labelling the classes for the labelled dataset. Therefore, the accuracy found in our model a model with the Louvain algorithm. As different networks have different community structure, generating the labelled dataset is necessary for each individual network. If we have sufficient labelled data available for a network, we can skip the initial label generation for the networks using Louvain algorithm.

As we are using semi-supervised classification, we divide our dataset into training, validation and test sets. We keep the ratio of training data lower for semi-supervised learning. Initially, we use a random
distribution of training, validation and test sets. We keep the label rate 0.001 for the random distribution. Label rate is the ratio of the size of training data to total labelled data. Validation set and test set are divided into 1 : 2 ratio within the rest of the labelled data. Random distribution does not work well because many classes might have very few labelled instances and those classes cannot be classified correctly. Thus, we choose classes that have at least 30 instances and discard classes having fewer instances than 30. We name it as ‘30-I’ for 30 instances and ‘N-I’ in general while considering classes with N instances. Training, validation and test sets are divided similarly as random distribution. The main difference is that instead of the whole labelled dataset, instances are chosen depending on the number of instances in each class. The performance of node-based feature set is poor using ‘30-I’ (we describe the reason in Section 2.6). Therefore, we choose ‘40-I’ for this feature set and get improved accuracy. Based on the number of labelled data using ‘30-I’ and ‘40-I’, we choose the optimal label rate given in Section 2.6.

Algorithm 15: GCN using Mini-batch Gradient Descent

Input: Graph Adjacency Matrix $A$, Feature Matrix $X$, Label $Y$, Layer Depth $K$;
Output: Node representation $Z$

1. Partition graph nodes into $b$ mini-batches $V_1, V_2, \cdots, V_B$

2. for $k = 1 \ldots K$ do
   3. for $b = 1 \ldots B$ do
      4. for $u \in V_b$ do
         5. Update Weight matrix $W^{(k)}$
      6. end
   7. end
   8. Calculate $Z$
9. end

Table 5.3: Networks Used in Experimentation

<table>
<thead>
<tr>
<th>Network Type</th>
<th>Name</th>
<th>Vertices</th>
<th>Edges</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Citation</td>
<td>cit-DBLP</td>
<td>12,591</td>
<td>49,743</td>
<td>DBLP Citation Network</td>
<td>[142]</td>
</tr>
<tr>
<td></td>
<td>cit-hepTh</td>
<td>27,770</td>
<td>352,807</td>
<td>Physics Theory paper citation network</td>
<td>[141]</td>
</tr>
<tr>
<td></td>
<td>cit-hepPh</td>
<td>34,546</td>
<td>421,578</td>
<td>Physics Phenomology paper citation network</td>
<td>[141]</td>
</tr>
<tr>
<td>Type</td>
<td>Data Source</td>
<td>EDG</td>
<td>Vertices</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------------</td>
<td>-----------</td>
<td>-------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication</td>
<td>email-Eu-core</td>
<td>1,005</td>
<td>25,571</td>
<td>Email data from a large European research institution [141]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>email-Enron</td>
<td>36,692</td>
<td>183,831</td>
<td>Email communication network from Enron [141]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>email-dnc</td>
<td>1,900</td>
<td>37,400</td>
<td>2016 Democratic National Committee email leak [142]</td>
<td></td>
</tr>
<tr>
<td>Internet</td>
<td>p2p-Gnutella31</td>
<td>62,586</td>
<td>147,892</td>
<td>Gnutella peer to peer network from August 31 2002 [141]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>p2p-Gnutella24</td>
<td>26,518</td>
<td>65,369</td>
<td>Gnutella peer to peer network from August 24 2002 [141]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>p2p-Gnutella08</td>
<td>6,301</td>
<td>20,777</td>
<td>Gnutella peer to peer network from August 8 2002 [141]</td>
<td></td>
</tr>
<tr>
<td>Collaboration</td>
<td>DBLP</td>
<td>317,080</td>
<td>1,049,866</td>
<td>DBLP collaboration network [141]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ca-GrQc</td>
<td>5,242</td>
<td>14,496</td>
<td>Collaboration network of Arxiv General Relativity [141]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ca-HepPh</td>
<td>12,008</td>
<td>118,521</td>
<td>Collaboration network of Arxiv High Energy Physics [141]</td>
<td></td>
</tr>
<tr>
<td>Biological</td>
<td>human-BNU</td>
<td>177,679</td>
<td>15,669,037</td>
<td>Human brain network where edges represent fiber tracts that connect one vertex to another [142]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bio-grid-human</td>
<td>9,527</td>
<td>62,364</td>
<td>category of Biological Networks [142]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bio-human-gene-2</td>
<td>22,283</td>
<td>12,345,963</td>
<td>Human gene regulatory network derived from analyzing gene expression profiles [142]</td>
<td></td>
</tr>
<tr>
<td>Road</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>----------</td>
<td>----------</td>
<td>--------------------------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>road-euroroad</td>
<td>1,174</td>
<td>1,417</td>
<td>Europe Road Network</td>
<td>[142]</td>
<td></td>
</tr>
<tr>
<td>road-usroads</td>
<td>129,164</td>
<td>165,435</td>
<td>US road network</td>
<td>[142]</td>
<td></td>
</tr>
<tr>
<td>road-luxembourg-osm</td>
<td>114,599</td>
<td>119,666</td>
<td>Open Street Map Road</td>
<td>[142]</td>
<td></td>
</tr>
<tr>
<td>slash-dot</td>
<td>77,360</td>
<td>905,468</td>
<td>Slashdot social network, containing friend/foe links between the users of Slashdot, obtained in November 2008</td>
<td>[141]</td>
<td></td>
</tr>
<tr>
<td>wiki-Vote</td>
<td>7,115</td>
<td>103,689</td>
<td>Wikipedia who-votes-on-whom network</td>
<td>[141]</td>
<td></td>
</tr>
<tr>
<td>ego-Facebook</td>
<td>4,039</td>
<td>88,234</td>
<td>Social circles from Facebook (anonymized)</td>
<td>[141]</td>
<td></td>
</tr>
</tbody>
</table>

We apply the GCN Model as described in [54] for training. We use non-linear activation function ReLU [172] for propagation rule. We perform forward propagation through the GCN. We apply the ReLU function row-wise on the last layer in the GCN. We compute the cross entropy loss on known node labels. We backpropagate the loss and update the weight matrices in each layer. The model is trained for a specified number of epochs where the loss is calculated for each training example and the error is backpropagated. But in case of larger networks (networks having more than 70,000 vertices), instead of Batch Gradient Descent, we choose Mini-batch Gradient Descent. Batch size of 1200 is used throughout the experimentation. We sample the mini batches by selecting the nodes with dense neighborhoods. We keep the track of the nodes in consecutive two layers so that our algorithm does not suffer from sparse connection problem. Our Mini-batch GCN algorithm is presented in Algorithm 15. The next step is then invoked to update the model parameters. We train our model to a maximum number of epoch and use early stopping with a window size of 10. We stop training if the validation loss does not decrease for 10 consecutive epochs. We tune the hyper parameters of the model to get the best performance. The optimized parameters are described in Section 2.6. These parameters are dependent on the nature and properties of the network.

To speed up the execution of our model, we use the functionality of PyTorch on GPUs using CUDA.
Table 5.4: Performance Evaluation for Different Networks Based on Identity Matrix (I.M.) and Graph (G.) Features

<table>
<thead>
<tr>
<th>No</th>
<th>Network</th>
<th>Accuracy I.M.</th>
<th>Accuracy G.</th>
<th>Precision I.M.</th>
<th>Precision G.</th>
<th>Recall I.M.</th>
<th>Recall G.</th>
<th>F1 Score I.M.</th>
<th>F1 Score G.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cit-DBLP</td>
<td>85.44</td>
<td>69.17</td>
<td>0.81</td>
<td>0.61</td>
<td>0.87</td>
<td>0.54</td>
<td>0.84</td>
<td>0.57</td>
</tr>
<tr>
<td>2</td>
<td>cit-hepTh</td>
<td>84.54</td>
<td>57.93</td>
<td>0.83</td>
<td>0.53</td>
<td>0.81</td>
<td>0.64</td>
<td>0.82</td>
<td>0.58</td>
</tr>
<tr>
<td>3</td>
<td>cit-hepPh</td>
<td>82.80</td>
<td>65.60</td>
<td>0.80</td>
<td>0.66</td>
<td>0.87</td>
<td>0.50</td>
<td>0.83</td>
<td>0.57</td>
</tr>
<tr>
<td>4</td>
<td>email-Eu-core</td>
<td>86.40</td>
<td>60.24</td>
<td>0.85</td>
<td>0.67</td>
<td>0.82</td>
<td>0.54</td>
<td>0.83</td>
<td>0.60</td>
</tr>
<tr>
<td>5</td>
<td>email-Enron</td>
<td>82.39</td>
<td>56.54</td>
<td>0.86</td>
<td>0.55</td>
<td>0.75</td>
<td>0.55</td>
<td>0.80</td>
<td>0.55</td>
</tr>
<tr>
<td>6</td>
<td>email-dnc</td>
<td>85.49</td>
<td>60.72</td>
<td>0.84</td>
<td>0.64</td>
<td>0.75</td>
<td>0.65</td>
<td>0.80</td>
<td>0.65</td>
</tr>
<tr>
<td>7</td>
<td>p2p-Gnutella31</td>
<td>86.35</td>
<td>61.30</td>
<td>0.86</td>
<td>0.65</td>
<td>0.79</td>
<td>0.50</td>
<td>0.83</td>
<td>0.57</td>
</tr>
<tr>
<td>8</td>
<td>p2p-Gnutella24</td>
<td>84.61</td>
<td>55.55</td>
<td>0.85</td>
<td>0.53</td>
<td>0.80</td>
<td>0.68</td>
<td>0.82</td>
<td>0.60</td>
</tr>
<tr>
<td>9</td>
<td>p2p-Gnutella08</td>
<td>82.39</td>
<td>62.50</td>
<td>0.82</td>
<td>0.64</td>
<td>0.87</td>
<td>0.65</td>
<td>0.84</td>
<td>0.65</td>
</tr>
<tr>
<td>10</td>
<td>DBLP</td>
<td>82.10</td>
<td>69.02</td>
<td>0.85</td>
<td>0.54</td>
<td>0.77</td>
<td>0.79</td>
<td>0.81</td>
<td>0.64</td>
</tr>
<tr>
<td>11</td>
<td>ca-GrQc</td>
<td>85.80</td>
<td>61.58</td>
<td>0.84</td>
<td>0.51</td>
<td>0.87</td>
<td>0.79</td>
<td>0.85</td>
<td>0.62</td>
</tr>
<tr>
<td>12</td>
<td>ca-HepPh</td>
<td>82.95</td>
<td>67.74</td>
<td>0.82</td>
<td>0.63</td>
<td>0.87</td>
<td>0.79</td>
<td>0.84</td>
<td>0.70</td>
</tr>
<tr>
<td>13</td>
<td>human-BNU</td>
<td>85.10</td>
<td>60.97</td>
<td>0.81</td>
<td>0.58</td>
<td>0.76</td>
<td>0.70</td>
<td>0.79</td>
<td>0.63</td>
</tr>
<tr>
<td>14</td>
<td>bio-grid-human</td>
<td>83.20</td>
<td>68.16</td>
<td>0.86</td>
<td>0.61</td>
<td>0.79</td>
<td>0.66</td>
<td>0.82</td>
<td>0.63</td>
</tr>
<tr>
<td>15</td>
<td>bio-human-gene-2</td>
<td>86.90</td>
<td>55.43</td>
<td>0.85</td>
<td>0.63</td>
<td>0.78</td>
<td>0.67</td>
<td>0.81</td>
<td>0.65</td>
</tr>
<tr>
<td>16</td>
<td>road-euroad</td>
<td>83.42</td>
<td>61.96</td>
<td>0.85</td>
<td>0.69</td>
<td>0.77</td>
<td>0.75</td>
<td>0.81</td>
<td>0.72</td>
</tr>
<tr>
<td>17</td>
<td>road-usroads</td>
<td>81.14</td>
<td>69.81</td>
<td>0.83</td>
<td>0.52</td>
<td>0.81</td>
<td>0.75</td>
<td>0.82</td>
<td>0.61</td>
</tr>
<tr>
<td>18</td>
<td>road-luxembourg-osm</td>
<td>81.96</td>
<td>59.11</td>
<td>0.86</td>
<td>0.55</td>
<td>0.76</td>
<td>0.75</td>
<td>0.81</td>
<td>0.64</td>
</tr>
<tr>
<td>19</td>
<td>slash-dot</td>
<td>86.35</td>
<td>65.32</td>
<td>0.84</td>
<td>0.69</td>
<td>0.80</td>
<td>0.65</td>
<td>0.82</td>
<td>0.67</td>
</tr>
<tr>
<td>20</td>
<td>wiki-Vote</td>
<td>85.26</td>
<td>58.76</td>
<td>0.86</td>
<td>0.60</td>
<td>0.81</td>
<td>0.68</td>
<td>0.83</td>
<td>0.64</td>
</tr>
<tr>
<td>21</td>
<td>ego-Facebook</td>
<td>83.76</td>
<td>63.85</td>
<td>0.86</td>
<td>0.59</td>
<td>0.81</td>
<td>0.69</td>
<td>0.83</td>
<td>0.64</td>
</tr>
</tbody>
</table>

Pytorch uses one GPU by default and we use multiple GPUs by running our model parallely using `DataParallel` approach. `DataParallel` automatically splits data and sends job orders to multiple models on multiple GPUs. After each model finishes their job, `DataParallel` collects and merges the results before returning. As available GPU memory is limited to 5.566GB in QB2, while working with the larger networks, we load a chunk (portion) of data since the entire graph does not fit in memory for computation. We keep the chunk size 1200 in our experimentation. Therefore, for our largest (according to number of vertices) graph [DBLP (317,080 vertices)], around 1.37GB space is taken by each of the adjacency matrix and the identity matrix. So, for a single pass of the full network, it takes around 70,000 iterations for loading the data into memory.
for further computations. We also change our model from Batch Gradient Descent to Mini-batch Gradient Descent in GPU because full training set is not available to the model for larger datasets. Thus, the weights are updated using the mean gradient of the mini-batch.

5.1.5 Experimental Evaluation

In this section, we summarize the results from our experimentation. We describe the datasets used in our experimentation in Sec. 5.1.5.1, and report the computational setting in Sec. 5.1.5.2 we have used for this paper. In Sec. 5.1.5.3, we describe the results to optimize our model with minimized error. We compare our model with the state-of-the-art methods in Sec. 5.1.5.4. Finally, in Sec. 5.1.5.5, we show the performance of our model in different computing environment.

5.1.5.1 Dataset

We describe the datasets used in our experiments in Table 5.3. All of these networks are taken from 7 different domains. All of these are real-world networks and have been collected from SNAP [141] and Network Repository [142]. Experimenting on a large range of networks gives us the chance to evaluate our method extensively, on datasets with varied topological and structural characteristics.

5.1.5.2 Experimental setting

Some initial experiments on smaller graphs have been conducted on Ubuntu 16.04 LTS, intel core i7 processor @3.4GHz×8, and 16GB memory. However, we perform all other experiments (including the ones on large graphs) on Louisiana Optical Network Infrastructure (LONI). QB2 [129], a 1.5 Petaflop peak performance cluster containing 504 compute nodes with over 10,000 Intel Xeon processing cores of 2.8 GHz has been used for the experiments performed. QB2 has two 960 NVIDIA Tesla K20x GPUs with 128GB memory. QB2 has RedHat Enterprise Linux 6 Operating System, 56 Gb/sec (FDR) InfiniBand 2:1 oversubscribed mesh, 1 Gb/sec Ethernet management network, 10 Gb/sec and 40 Gb/sec external connectivity. NVIDIA Driver 396.51 has been used with CUDA 10.0.

5.1.5.3 Model Optimization

We summarize our result in Table 5.4 with respect to the model performance evaluation metrics. We consider the modularity-based communities calculated using the Louvain algorithm [154, 158] as ground-truth. Thus, the accuracy of our model compares our algorithm with the Louvain algorithm. In addition to accuracy, we also measure precision, recall, and F1 score as these metrics are widely used for multi-class classification.
Table 5.5: Label Rate of Different Networks Used in Experiments for Both Identity Matrix Feature Set and Graph Feature Set

<table>
<thead>
<tr>
<th>Network</th>
<th>Identity Matrix</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>cit-DBLP</td>
<td>0.005</td>
<td>0.018</td>
</tr>
<tr>
<td>cit-hepTh</td>
<td>0.005</td>
<td>0.011</td>
</tr>
<tr>
<td>cit-hepPh</td>
<td>0.014</td>
<td>0.006</td>
</tr>
<tr>
<td>email-Eu-core</td>
<td>0.002</td>
<td>0.010</td>
</tr>
<tr>
<td>email-Enron</td>
<td>0.010</td>
<td>0.011</td>
</tr>
<tr>
<td>email-dnc</td>
<td>0.017</td>
<td>0.004</td>
</tr>
<tr>
<td>p2p-Gnutella31</td>
<td>0.025</td>
<td>0.007</td>
</tr>
<tr>
<td>p2p-Gnutella24</td>
<td>0.003</td>
<td>0.021</td>
</tr>
<tr>
<td>p2p-Gnutella08</td>
<td>0.007</td>
<td>0.023</td>
</tr>
<tr>
<td>DBLP</td>
<td>0.019</td>
<td>0.010</td>
</tr>
<tr>
<td>ca-GrQc</td>
<td>0.003</td>
<td>0.016</td>
</tr>
<tr>
<td>ca-HepPh</td>
<td>0.002</td>
<td>0.020</td>
</tr>
<tr>
<td>human-BNU</td>
<td>0.002</td>
<td>0.002</td>
</tr>
<tr>
<td>bio-grid-human</td>
<td>0.007</td>
<td>0.006</td>
</tr>
<tr>
<td>bio-human-gene-2</td>
<td>0.012</td>
<td>0.005</td>
</tr>
<tr>
<td>road-euroroad</td>
<td>0.009</td>
<td>0.019</td>
</tr>
<tr>
<td>road-usroads</td>
<td>0.007</td>
<td>0.016</td>
</tr>
<tr>
<td>road-luxembourg-osm</td>
<td>0.019</td>
<td>0.007</td>
</tr>
<tr>
<td>slash-dot</td>
<td>0.003</td>
<td>0.006</td>
</tr>
<tr>
<td>wiki-Vote</td>
<td>0.009</td>
<td>0.024</td>
</tr>
<tr>
<td>ego-Facebook</td>
<td>0.013</td>
<td>0.008</td>
</tr>
</tbody>
</table>

problem. Precision is useful when the costs of false positive is high. Recall is useful when there is a high cost associated with false negative. F1 score is a better measure to use if we need to seek a balance between precision and recall and there is an uneven class distribution. We observe that identity matrix based feature set performs better than the node-based feature set. The reason behind is that the node-based features are not directly correlated with the communities. Two nodes connected to each other are not necessarily “close”. So, such node-based features are not a good predictor for communities. We get around 87% accuracy, 86% precision, 87% recall and 85% F1 score for identity matrix based feature set. In addition, for node-based feature set, we get around 70% accuracy, 69% precision, 79% recall and 72% F1 score. Imbalanced labelled data might be a reason for the reduced accuracy. The number of classes for multi-class classification is very large. It is one of the reasons for reduced performance of the model in terms of accuracy. In order to get the best model, we tuned the hyper-parameters. For brevity, we have shown the experimentation for the identity matrix based feature set only. For node-based feature set, we use the same optimized hyper parameters.
Table 5.6: Performance Evaluation for Different Networks with Random Train-Test Distribution [I.M.-Identity Matrix, G.-Node-based Graph features]

<table>
<thead>
<tr>
<th>Network</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1 Score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I.M.</td>
<td>G.</td>
<td>I.M.</td>
<td>G.</td>
</tr>
<tr>
<td>ego-Facebook</td>
<td>55.18</td>
<td>30.33</td>
<td>.45</td>
<td>.19</td>
</tr>
<tr>
<td>email-Eu-core</td>
<td>66.26</td>
<td>55.48</td>
<td>.55</td>
<td>.26</td>
</tr>
<tr>
<td>ca-GrQc</td>
<td>63.30</td>
<td>13.17</td>
<td>.70</td>
<td>.10</td>
</tr>
</tbody>
</table>

5.1.5.3.1 Training Data Distribution Distribution of training set and test set plays an important role to improve the accuracy of the model. Initially, we use a random distribution of training and test set. Random distribution does not work well in this case. Table 5.6 shows that the accuracy is very poor for some of the graphs using both identity matrix feature as well as node-based graph feature.

Table 5.7: Model Performance Improvement Using Changed Number of Labels Per Class for Node-based Graph Features

<table>
<thead>
<tr>
<th>Network</th>
<th>30-I</th>
<th>40-I</th>
</tr>
</thead>
<tbody>
<tr>
<td>cit-DBLP</td>
<td>47.88</td>
<td>69.17</td>
</tr>
<tr>
<td>email-Eu-core</td>
<td>55.45</td>
<td>60.24</td>
</tr>
<tr>
<td>p2p-Gnutella08</td>
<td>49.35</td>
<td>62.50</td>
</tr>
<tr>
<td>ca-GrQc</td>
<td>25.69</td>
<td>61.58</td>
</tr>
<tr>
<td>bio-grid-human</td>
<td>36.76</td>
<td>68.16</td>
</tr>
<tr>
<td>road-euroroad</td>
<td>43.86</td>
<td>61.96</td>
</tr>
<tr>
<td>ego-Facebook</td>
<td>35.25</td>
<td>63.85</td>
</tr>
</tbody>
</table>

Thus, instead of random distribution, we choose ‘30-I’ described in Section 2.4. For Identity Matrix Feature Set, the model accuracy shown in Table 5.4 is found using ‘30-I’ classification. In case of Graph Feature set, initially we consider ‘30-I’ but find that accuracy is poor. Later we choose ‘40-I’ and accuracy improves on an average 69% given in Table 5.7. Choosing ‘40-I’ suffers from the exclusion of many mid-sized communities from the dataset. Results shown in Table 5.4 for node-based graph feature set is given for ‘40-I’ classification. The improved accuracy is in between 55.4% to 69.8% for all networks reflected in Table 5.4.

5.1.5.3.2 Label Rate Label rate is an important parameter to evaluate the model performance. We keep the label rate small to prevent the model from over-fitting. Figure 5.1 depicts the change of accuracy with varying label rate for identity matrix based feature set. Networks from the same domain show similar behavior. To keep it simple, we have shown a single network from each domain in Figure 5.1. For social, road and biological network domains, the optimal label rate is in between 0.001 to 0.01. The model tends to over-fit if the label rate goes beyond 0.025 and above. For the rest of the network domains, the optimal label rate is in between 0.01 to 0.025. Based on the optimal values, we use the label rate for each network given
Figure 5.1: Change of accuracy with increasing label rate for identity matrix based feature set. The optimal label rate is in between the range 0.001 to 0.025 for all domains of network, to prevent the model from over-fitting.

in Table 5.5. Number 1−21 refers to the networks described in Table 5.4. For node-based graph feature set, we keep the label rate within the range 0.001 to 0.025, to prevent the model from over-fitting.

Table 5.8: Change of Model Performance with Varying Number of Convolutional Layers

<table>
<thead>
<tr>
<th>Network</th>
<th>2 Layers</th>
<th>3 Layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>cit-DBLP</td>
<td>85.44</td>
<td>62.34</td>
</tr>
<tr>
<td>email-Eu-core</td>
<td>86.40</td>
<td>59.46</td>
</tr>
<tr>
<td>p2p-Gnutella08</td>
<td>82.39</td>
<td>39.54</td>
</tr>
<tr>
<td>ca-GrQc</td>
<td>85.80</td>
<td>65.12</td>
</tr>
<tr>
<td>bio-grid-human</td>
<td>83.20</td>
<td>35.26</td>
</tr>
<tr>
<td>road-euroroad</td>
<td>83.42</td>
<td>55.52</td>
</tr>
<tr>
<td>ego-Facebook</td>
<td>83.76</td>
<td>64.34</td>
</tr>
</tbody>
</table>

5.1.5.3.3 Number of Convolutional Layers  Increasing the number of convolutional layers does not serve our purpose well and the model accuracy decreases by 35.5% on average given in Table 5.8. So, we use a 2-layer GCN for our model in all experiments.
Figure 5.2: Accuracy for different networks varying learning rate hyper parameter. For learning rate of 0.01, best accuracy is achieved for all network domains.
Figure 5.3: Accuracy for different networks varying dropout hyper parameter. For Citation, Communication and Internet Networks optimum dropout is 0.5. Except Road Networks, other domains have 0.1 optimum dropout.
5.1.5.3.4 Learning Rate  Figure 5.2 shows the change of accuracy with learning rate for different types of networks. With smaller learning rate of 0.01, we achieve the best accuracy for each of the networks. We have increased the learning rate up to 0.1 and have not showed results for higher values because the accuracy drops a lot beyond this value. We separately plot each types of networks to understand their pattern. All of these network categories follow a similar pattern. In case of biological networks (Figure 5.2e), after reduced accuracy, the accuracy does not go beyond 80%, and learning rate can be relaxed to 0.1. The change of accuracy is also low and varies within the range of 4 to 5. For communication (Figure 5.2b) and social (Figure 5.2g) networks the range of change of accuracy is between 4 to 7. In case of the rest types of networks (Figures 5.2a, 5.2c, 5.2d, 5.2f), the change of accuracy is higher and varies in between 6 to 11.

5.1.5.3.5 Dropout  We also vary the dropout given in Figure 5.3 and determine the optimized parameter with maximum accuracy. For collaboration (Fig. 5.3d), biological (Fig.5.3e) and social (Fig. 5.3g) networks, accuracy decreases with increasing dropout. In case of road networks (Fig.5.3f), accuracy decreases with dropout. But for “road-euroroad”, accuracy is increased up to a certain point, 0.5 then starts decreasing. For the rest three types of networks (Fig.5.3a, 5.3b, 5.3c), accuracy increases with dropout until 0.5, then
Figure 5.5: Accuracy for different networks varying the number of epoch. The optimal number of epoch for smaller, medium and larger networks are 200 to 250, 500 to 650 and 700 to 800 respectively.

accuracy falls. Increasing the dropout lessens the run-time of the model shown in 5.4. In certain cases, where we can relax the accuracy to some extent, we can opt for a higher dropout.

5.1.5.3.6 Number of Epoch We also optimize the number of epoch for each graph that gives the best performance in terms of accuracy, as depicted in Figure 5.5. From Figure 5.5a, we observe that for relatively smaller sized networks, the optimum number of epoch is 200 – 250. In case of the medium sized networks, the epoch size varies in between 500 to 650, as shown in Figure 5.5b. For larger networks, keeping the epoch in between 700 – 800 provides the best accuracy, as shown in Figure 5.5c.
5.1.5.4 Comparison with the Related Methods

Table 5.9: Comparison of Our GCN-based Method and Other Related Existing Methods [4, 5] on arxiv and PPI datasets.

<table>
<thead>
<tr>
<th>Method</th>
<th>arxiv Dataset [Performance Metric: Test Accuracy (%)]</th>
<th>PPI Dataset [Performance Metric: F1 Score]</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLP</td>
<td>55.50±0.23</td>
<td>N/A</td>
</tr>
<tr>
<td>Node2Vec</td>
<td>70.07±0.13</td>
<td>N/A</td>
</tr>
<tr>
<td>GCN</td>
<td>71.74±0.29</td>
<td>N/A</td>
</tr>
<tr>
<td>GraphSAGE</td>
<td>71.19±0.27</td>
<td>0.612</td>
</tr>
<tr>
<td>Our Model</td>
<td>71.43±0.17</td>
<td>0.698</td>
</tr>
</tbody>
</table>

To compare our model with other existing algorithms, we perform experiments on the same datasets the other papers have used. Although our identity matrix based feature set shows better performance, we use the same feature set used by other methods. We do not use identity matrix based feature set to make the comparison consistent and fair. Splitting of the training, validation and test sets are also done in the same manner. arxiv is a citation network with 169,343 nodes and 1,166,243 edges [4]. Our model shows comparable performance with the base GCN [54] and GraphSAGE [5] shown in Table 5.9. We also test our model on another dataset PPI [5], a biological network with 56,944 nodes and 818,716 edges. F1 Score has been reported in [5]. We get better F1 Score given in Table 5.9. We use Mini-batch Gradient Descent for our model for the performance comparison for both datasets.

5.1.5.5 Model Performance Evaluation

We choose the largest graphs from each types of networks and evaluate their performance in different computing environment. As, node-based graph feature set does not perform better compared to the identity matrix feature set, we show the model performance for the identity matrix only. For the largest graph in terms of vertices, with 0.3 Million nodes, the model takes around 3.7 hrs to run on CPUs whereas around 1 hr on GPUs. In case of the largest graph considering edges (15.7 Million), the time required on CPUs is 1.1 hours and 0.7 hours on GPUs. While considering the experiments done on CPUs, we have used a single node of the computing cluster comprising of 20 processing cores. No parallel mechanism has been applied for the CPU computations. From Figure 5.6 we see that we get around 2.5× speedup using 1 GPU and 4× speedup using 2 GPUs over CPUs.
Figure 5.6: Speed-up of our model for best performance on GPUs over CPUs. 4× speed-up is observed using 2 GPUs over CPUs.

5.1.6 Conclusion

Semi-supervised learning for community detection is very useful for large-scale networks where detecting communities is computationally expensive and we have access to a few labelled data. Based on the limited information, we can detect communities of the full network via semi-supervised learning. Our semi-supervised classification model shows an accuracy of up to 86.9%, and 0.85 F1 score for identity matrix based feature set. Node-based graph feature set does not perform equally as the identity matrix based feature set but can be used when the memory is limited and the accuracy of the model can be relaxed a bit. We make our model scalable for larger datasets using Mini-batch Gradient Descent resolving the memory issue of GCN. We experiment on 7 different domains of real-world networks and see how our model performs for each of the domains. We have included large-scale datasets in our experiments compared to the smaller datasets used by other related work in literature. We compare our model with the state-of-the-art methods and achieve comparable performance. Our model shows 4× speedup using 2 GPUs over CPUs.
5.2 Data Parallel Large Sparse Deep Neural Network on GPU

Sparse Deep Neural Network (DNN) is an emerging research area since deploying deep neural networks with limited resources is very challenging. In this work, we provide a scalable solution to the Sparse DNN Challenge—a challenge posed by MIT/IEEE/Amazon GraphChallenge.org—by designing data parallelism on GPUs. We provide a solution based on Python TensorFlow as it is a widely used tool in different scientific applications for deep learning. We use the datasets provided by GraphChallenge, derived from the MNIST handwritten letters. We use the Synthetic DNNs from RadiX-Net with varying number of neurons and layers. We implement a data parallel implementation of Sparse DNN using TensorFlow on GPU. Our solution shows up to $4.7 \times$ speedup over the baseline serial MATLAB implementation given in GraphChallenge. In addition to that, our TensorFlow GPU implementation demonstrates a 3-fold speedup over our TensorFlow CPU implementation.

5.2.1 Introduction

Nowadays, we have been observing a rapid growth of deep learning applications in business, scientific and technical domains [173, 82, 174, 175]. Such applications are assumed to reign our lives in near future. For example, deep learning plays an important role in the field of speech recognition, visual object recognition, object detection, drug discovery, and genomics [176]. To address important problems in data-intensive computing [173], e.g., extracting complex patterns from massive volumes of data, semantic indexing, sentiment analysis, data tagging [82], fast information retrieval [174, 175], and simplifying discriminative tasks [177], deep learning techniques are being utilized heavily. However, deep learning models these days require a significant amount of memory and computing power which become a bottleneck in the conditions where such resources are limited [178]. Deep Neural Networks (DNNs) are often much harder to train than shallow neural networks. Larger neural networks often perform better because larger number of layers/features allow more non-linear boundaries. However, such larger networks are constrained by large memory requirements. Therefore, large-scale optimization [108, 179, 180, 181, 182] is needed to address such challenges [183, 184]. Sparse (pruned) neural networks deliver comparable performance with less amount of memory resources. In order to use machine learning features on mobile devices (facial recognition or voice assistants for instance), one needs small neural networks. Such on-the-go applications are another important motivation for using Sparse DNNs. With an increasing number of layers and neurons, the weight matrices can be made sparse to tackle the memory limitations. Pruning [55] results in better generalisation results, improved speed of processing the results and a reduced size as well. The need for sparse DNNs has inspired the Sparse Deep
Neural Network Graph Challenge by MIT/IEEE/Amazon GraphChallenge community. The GraphChallenge seeks contribution from community of researchers to come up with solutions in the field of sparse data and graph analytics [114, 111, 185] by collecting data from different scientific domains [186]. Sparse Deep Neural Network Graph Challenge performs neural network inference on a variety of sparse deep neural networks.

In this study, we present a solution to the Sparse DNN Graph Challenge using Python TensorFlow. We achieve up to 4.7-fold speedup using GPUs over the serial MATLAB implementation provided in GraphChallenge by implementing Sparse DNN in data parallel mode. We also compare different strategies of Distributed Training API of TensorFlow and demonstrate their performance empirically.

The rest of this paper is organized as follows. We describe the related works in Section 5.2.3. Background of Deep Neural Networks and the parallelization techniques are discussed in Section 5.2.2. In Section 5.2.4, we describe our methods to implement a TensorFlow based data parallel sparse DNN using different strategies of distributed training of TensorFlow in CPU as well as GPU. Our datasets and experimentation environment are described in Section 5.2.5. A detailed analysis of the results is presented in Section 5.2.6. Finally, Section 5.2.7 summarizes the findings and concludes the paper with a discussion of future possibilities.

5.2.2 Preliminaries

In this section, we discuss the basic concepts of DNN and the scope for parallelization of our serial baseline methodology.

5.2.2.1 How DNNs work

A sample DNN is illustrated in Fig. 5.7. The primary mathematical operation performed by a DNN is the inference, or forward propagation step. Inference is executed repeatedly during training to determine both the weight matrix $W^{(l)}$ and the bias vectors $b^{(l)}$ of the DNN. The inference computation is given by Equation 5.3.

$$y^{(l+1)} = h(y^{(l)}W^{(l)} + b^{(l)})$$ (5.3)

where, $h()$ is a nonlinear function applied to each element of the vector. A commonly used function is the rectified linear unit (ReLU) given by Equation 5.4. For the Sparse DNN challenge, $h()$ also has an upper
Figure 5.7: Four layer ($L = 4$) deep neural network architecture. The input features $y^{(0)}$ are passed through a series of network layers $W^{(l=0,1,2,3)}$ with bias terms $b^{(l=0,1,2,3)}$, that produce scores for categories $y^{(L=4)}$ limit set to 32. ReLU function sets negative values to 0 and values greater than 32 to 32.

$$h(y) = \begin{cases} y & \text{if } 0 < y < 32 \\ 0 & \text{if } y < 0 \\ 32 & \text{if } y > 32 \end{cases} \quad (5.4)$$

$W(i,j) > 0$ implies a connection between neuron $i$ and neuron $j$, using the standard graph community convention. Dimension of $W^{(l)}$ matrix is $k \times k$ where, $k$ denotes the number of neurons.

The quantities $y^{(l)}$ are row vectors. Each row is a feature vector. In Sparse DNN Challenge, according
to standard graph community terminology, left matrix multiply is used to progress through the network. Transposing all matrices and multiplying on the right can be used according to Standard AI definitions. The quantity $b^{(l)}$ is a bias row vector applied to each input.

When training a DNN, or performing inference on many different inputs, it is usually necessary to compute multiple $y^{(l)}$ vectors at once in a batch that can be denoted as the matrix $Y^{(l)}$. The dimension of matrix $Y^{(l)}$ is $n \times p$ where, $n =$ number of inputs, $p =$ number of features. In matrix form, the inference step follows Equation 5.5.

$$Y^{(l+1)} = h(Y^{(l)}W^{(l)} + B^{(l)}) \quad (5.5)$$

Here, $B^{(l)}$ is a replication of $b^{(l)}$ along columns given by Equation 5.6.

$$B^{(l)} = b^{(l)}|Y^{(l)}one\rangle_0 \quad (5.6)$$

Here, $one$ is a column array of 1’s, and $\| \rangle_0$ is the zero norm. The overall steps to calculate inference is given in Algorithm 16.

---

**Algorithm 16: Serial Inference Calculation in MATLAB**

**Data:** $W$ is the Weight Matrix, bias is the Bias Vector, $Y0$ is the Input, MNIST Image

**Result:** $Y$

1. $Y_{max} \leftarrow 32$
2. $Y \leftarrow Y0$
3. $nlayers \leftarrow length(W)$
4. **for** layer **until** nlayers **do**
5.   $Z \leftarrow Y \times W\{layer\}$
6.   $b \leftarrow bias\{layer\}$
7.   $Y \leftarrow Z + \text{bsxfun(@times, double(logical(Z)), bias\{layer\})}$ \hfill // \text{bsxfun(@x, a, b)} applies $x$ operation on arrays $a$ and $b$
8.   **if** $Y < 0$ **then**
9.       $Y \leftarrow 0$
10. **end**
11. **else if** $Y > Y_{max}$ **then**
12.       $Y \leftarrow Y_{max}$
13. **end**
14. **end**
5.2.2.2 Scopes for Parallelization

We aim at scaling up our sparse DNN based computing task through parallel implementation. To parallelize the DNN, we consider the following three possibilities.

5.2.2.2.1 Data Parallelism  In data parallelism [187], we split inputs across all devices. The weight matrices need to be replicated on every processor. This method might not be the most space efficient but the design of such method is comparatively less complex.

5.2.2.2.2 Model Parallelism  This paradigm requires splitting up the layers and running in a pipeline parallel mode. It requires communicating results after each group of layers. It saves memory but the design is more complex. Achieving balanced loads among multiple GPUs is very challenging. The complexity of different DNN layers varies, introducing significant efforts for programmers to partition model layers to GPUs in a balanced way. In order to achieve pipeline parallelism [16], understanding the mapping pattern of the communication is vital. Pipeline parallelism, itself, is a separate field of research. Another shortcoming of this method is to determine where the neural network is being split. The neural network requirements to process data also needs to be increased significantly. The weight staleness issue [17] is another concern. Since gradients are computed with stale weights, training instability and accuracy loss are persistent.

5.2.2.2.3 Hybrid Parallelism  This is a combination of data and model parallelism. Benefits from both of them can be combined in a hybrid [188, 189, 190] implementation.

In this work, we use data parallelism for our parallel implementation in an attempt to scale up our sparse DNN based task. We use TensorFlow Distributed Training to implement data parallelization for Sparse DNN in our experimentation. We have left model and hybrid parallelism as our future work.

5.2.3 Related Work

There exist several work in the current literature that focus on accelerating deep neural networks [58, 191, 56, 192, 193, 194, 195]. As provided by GraphChallenge, a serial algorithm for Sparse DNN is given in [191]. The MATLAB serial reference of the inference calculation is given in Algorithm 16. They present the serial timing measurements of the MATLAB code to be used as a benchmark. They also provide parallel implementation of the Sparse DNN benchmark. They parallelize the code by splitting feature vectors, then develop and test it on the MIT SuperCloud TX-Green supercomputer, Intel KNL processors with 192 GB RAM using pMatlab.
Another solution to the Sparse DNN Challenge has been provided by Davis et al. [56] using GraphBLAS [57]. The sequential performance of the GraphBLAS solution is $3 \times$ to $5 \times$ faster than the MATLAB reference implementation. OpenMP parallelism gives an additional $10 \times$ to $15 \times$ speedup on a 20-core Intel processor, $17 \times$ on an IBM Power8 system, and $20 \times$ on a Power9 system, for the largest problems. The performance metric, $Rate$ measures the throughput of the implementation as the ratio of the number of inputs times the number of connections in the DNN divided by the execution time [191]. There is an inconsistency in using the $Rate$ formula in serial MATLAB code and their GraphBLAS implementation. The discrepancy between the computed rate for MATLAB code in [56] and [191] can be misleading to the readers. This variation happens as the number of inputs is not included in the numerator for $Rate$ calculation in [56] for MATLAB code. Another Sparse DNN Challenge solution given in [58] shows a GPU implementation of the GraphBLAS standard. Their implementation shows a $1.94 \times$ speedup over the “SuiteSparse” CPU implementation of GraphBLAS.

Apart from the Sparse DNN challenge, the authors in [192] have proposed redundancy reduction schemes, including the software/hardware co-designs of the structured sparse neural network, an enhanced LRA algorithms and the ternary quantized gradients training for the distributed DNN. In [193], the authors propose a multiscale kernels approach to extract optimal criteria for saliency detection where they suppress nonsalient regions in the sparse labeling. The authors find an interesting way to connect nodes in a sparsely connected network [194]. Their sparse evolutionary training of artificial neural networks algorithm evolves an initial sparse topology (Erdős–Rényi random graph) of two consecutive layers of neurons into a scale-free topology during learning. Fully-connected layers of artificial neural networks are replaced with sparse ones before training and the parameters are reduced quadratically with no loss in accuracy. In [195], the authors develop a Structured Sparsity Learning (SSL) method to regularize the structures (i.e., filters, channels, filter shapes, and layer depth) of DNNs, that can learn a compact structure from a bigger DNN to reduce computation cost.

Our work is different from the work of [56, 58] as we focus on an efficient data parallel implementation using Python. Several deep learning frameworks, i.e., Tensorflow [59], PyTorch [60], are written in Python. Many scientific applications use these widely used deep learning frameworks. GraphBLAS focuses mainly on graph algorithms and is used by a specific community of graph researchers. Our work is generic in nature—it will help the end-users of different domains to apply these techniques with ease and achieve high performance capability.
Algorithm 17: A Pseudocode for creating custom layer to build our own model in TensorFlow

Data: \( W \) is the Weight Matrix, bias is the Bias Vector, \( Y_0 \) is the Input, MNIST Image

1. \( Y_{\text{max}} \leftarrow 32 \)

/* The custom layer class encapsulating both the layer's "weights" and a transformation from inputs to outputs */

2. class myClass(layers.Layer)
3.     __init__()
4.         // Input dimension and general initialization
5.     build()
6.         // Initialization of layer weights and bias
7.     call()
8.         // The layer's forward pass
9. {
10.     super.__init__()
11.     self.unit ← unit
12. }
13. build()
14. {
15.     self.w ← \( W_0 \)
16.     self.b ← \( \text{bias}_0 \)
17. }
18. call()
19. {
20.     \( Y \leftarrow \text{tf.matmul}(\text{inputs}, \text{self}.w) + \text{self.b} \)
21.     if \( Y < 0 \) then
22.         \( Y \leftarrow 0 \)
23.     end
24.     else if \( Y > Y_{\text{max}} \) then
25.         \( Y \leftarrow Y_{\text{max}} \)
26.     end
27.     return \( Y \)
28. }

5.2.4 Methodology

We use Python TensorFlow to solve the Sparse DNN problem. We implement our own layers by extending the tf.keras.Layer class and implementing: *\_\_init\_\_*, where we have done all input-independent initialization build. We know the shapes of the input tensors and can do the rest of the initialization call, doing the forward computation. For the activation function, we use the basic \text{matmul} function to implement our own ReLU function instead of the matrix multiplication given in Algorithm 16. A pseudocode for initialization of our custom layer is given in 17. We use this class myClass to build our model where the outer layer tracks the weights of the inner layer. For brevity, detailed code for model training is not described in the paper.

Initially, we start with a CPU implementation disabling CUDA. We divide the 60,000 samples into 60 chunks of 1000 samples/chunk. Later on, we activate a single GPU to check its performance. We use the
Central Storage Strategy from tf.distribute.Strategy API where all variables and operations will be placed on the GPU if a single GPU is available. This strategy is experimental and may change in future.

<table>
<thead>
<tr>
<th>Neuron per Layer</th>
<th>Layer</th>
<th>Total Neuron</th>
<th>Memory Required (GB)</th>
<th>Experiment Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,024</td>
<td>120</td>
<td>122,880</td>
<td>0.117</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>480</td>
<td>491,520</td>
<td>0.409</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>1,920</td>
<td>1,966,080</td>
<td>1.875</td>
<td>Yes</td>
</tr>
<tr>
<td>4,096</td>
<td>120</td>
<td>491,520</td>
<td>0.409</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>480</td>
<td>1,966,080</td>
<td>1.875</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>1,920</td>
<td>7,864,320</td>
<td>7.5</td>
<td>Only CPU, Exceed Local Memory for GPU</td>
</tr>
<tr>
<td>16,384</td>
<td>120</td>
<td>1,966,080</td>
<td>1.875</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>480</td>
<td>7,864,320</td>
<td>7.5</td>
<td>Only CPU, Exceed Local Memory for GPU</td>
</tr>
<tr>
<td></td>
<td>1,920</td>
<td>31,457,280</td>
<td>30</td>
<td>No</td>
</tr>
<tr>
<td>65,536</td>
<td>120</td>
<td>7,864,320</td>
<td>7.5</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>480</td>
<td>31,457,280</td>
<td>30</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>1,920</td>
<td>125,829,120</td>
<td>120</td>
<td>No</td>
</tr>
</tbody>
</table>

With multiple GPUs, we can use the extra computing power effectively by increasing the batch size. Using the largest batch size that fits the GPU memory utilizes the resources efficiently. But we have only 2 GPUs, limited by our system. So, we have used 60 chunks, batch size of 1000 throughout our experimentation.

For distributed training, we use tf.distribute.Strategy API. This is a TensorFlow API to distribute training across multiple GPUs, multiple machines or TPUs. We use the GPU functionality only. There are 6 different types of strategies available. But we choose to work with tf.distribute.MirroredStrategy because it is fully supported by Keras API and other strategies are still in experimental support phase. Mirrored Strategy supports synchronous distributed training on multiple GPUs on one machine. It creates one replica of the model per GPU device. Each variable in the model is mirrored across all the replicas. Together, these variables form a single conceptual variable called MirroredVariable. These variables are kept in synchronization with each other by applying identical updates. MirroredStrategy takes care of replicating the model’s training on the available GPUs, aggregating gradients, and more. Efficient all-reduce algorithms are used to communicate the variable updates across the devices. All-reduce aggregates tensors across all the devices by adding them up and makes them available on each device. It’s a fused algorithm that is very efficient and can reduce the overhead of synchronization significantly. There are many all-reduce algorithms and implementations available, depending on the type of communication available between devices. By default, it uses NVIDIA NCCL as the all-reduce implementation. The other communication schemes are
Hierarchical Copy All Reduce and Reduction To One Device. We try different communication schemes to achieve the best result.

Later on, we experiment with the strategy `tf.distribute.experimental.MultiWorkerMirroredStrategy` to perform a comparative analysis of different strategies in multiple GPUs. Similar to MirroredStrategy, it creates copies of all variables in the model on each device across all workers. Two different implementations are available in the experimental support: `CollectiveCommunication.NCCL` and `CollectiveCommunication.Ring` for collective operations. We use both and select the best one depending upon the number and kind of GPUs available in our system, and the network interconnect in the cluster.

We could not experiment with the 65536-neuron case as its storage requirement significantly exceeds the available memory of our system. Table 5.10 shows the memory requirement for working with each of the different sizes of DNNs. Scaling to a large dataset require a different approach, most certainly exploiting model parallelism or combination of model and data both, a hybrid strategy.
5.2.5 Experimental Setup

5.2.5.1 Environment

We used Louisiana Optical Network Infrastructure (LONI) to perform all the experiments. QB2 [129], a 1.5 Petaflop peak performance cluster containing 504 compute nodes with over 10,000 Intel Xeon processing cores of 2.8 GHz, 20 cores per node. A single node has two 960 NVIDIA Tesla K20x GPUs with 128GB memory, 500 GB HDD, that has been used for the experiments performed using GPUs. However, the available local memory for each GPU is limited to 5.566GB. QB2 has RedHat Enterprise Linux 6 Operating System, 56 Gb/sec (FDR) InfiniBand 2:1 oversubscribed mesh, 1 Gb/sec Ethernet management network, 10 Gb/sec and 40 Gb/sec external connectivity. NVIDIA Driver 396.51 has been used with CUDA 10.0 and the TensorFlow version is 1.14.

The serial MATLAB code has been run on an Intel Core i7-4770 CPU @ 3.4GHz ×8 processor and 16 GB RAM machine with MATLAB version R2015a.

5.2.5.2 Dataset

<table>
<thead>
<tr>
<th>Neurons</th>
<th>Size</th>
<th>Input Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>176 MB</td>
<td>32 × 32</td>
</tr>
<tr>
<td>4096</td>
<td>800 MB</td>
<td>64 × 64</td>
</tr>
<tr>
<td>16384</td>
<td>3.6 GB</td>
<td>128 × 128</td>
</tr>
</tbody>
</table>

Figure 5.9: Speedup of DNNs in Python TensorFlow CPUs and GPUs over the baseline MATLAB Implementation [L denotes number of Layers]

Sparse DNN Challenge requires input data or feature vectors $Y_0$. MNIST (Modified National Institute of Standards and Technology) is a large database of handwritten digits that is widely used for training and
testing DNN image processing systems. MNIST consists of 60,000 $28 \times 28$ pixel images. Truth categories for MNIST are included for performing inference using DNN with specific numbers of layers. The Sparse DNN Graph Challenge uses interpolated sparse versions of this entire corpus as input.

Sparse DNNs in MNIST corpus are resized to produce neural networks of varying dimensions shown in Table 5.11. Each $28 \times 28$ pixel image is resized to $32 \times 32$ (1024 neurons), $64 \times 64$ (4096 neurons), $128 \times 128$ (16384 neurons), and $256 \times 256$ (65536 neurons). The resized images are thresholded so that all values are either 0 or 1. The images are flattened into a single row to form a feature vector. The non-zero values are written as triples to a .tsv file where each row corresponds to a different image, each column is the nonzero pixel location and the value is 1. We also use Synthetic DNNs created using RadiX-Net [196] with varying number of neurons and layers.

129
5.2.6 Result

In this section, we describe the performance of our data parallel implementation of the Sparse DNN Challenge. While running the serial MATLAB implementation, we faced some difficulties because of the version compatibility issues of MATLAB. We needed to change some functions to run the baseline serial MATLAB code. We have summarized the experimentations performed on different Sparse DNNs in Tables 5.12 and 5.13. We compute speedup for our CPU and GPU implementations keeping the MATLAB sequential code as baseline. We get up to $4.7 \times$ speedup in terms of computational time over the baseline sequential MATLAB implementation using 2 GPUs. The inference rate changes with increasing computation power for various DNN sizes shown in Figure 5.8. Our change of inference rate is similar in nature as given by J. Kepner using pMatlab [191]. The runtime speedup over the baseline MATLAB for different DNNs is depicted in Figure 5.9. For different numbers of neurons, how the speedup varies with changing layers is portrayed. In case of smaller number of layers ($L=120$), all of the DNNs show speedup in between $4$ to $5$.

Besides, our parallel implementation using GPUs, our CPU sequential implementation is $1.34 \times -1.67 \times$ faster than the MATLAB sequential implementation. The speedup of our GPU implementation compared to our CPU implementation is shown in Figure 5.10. We get up to $3.38 \times$ speedup using 2 GPUs over the CPU implementation. The performance is proportionally scalable with the available number of GPUs. We can use only 2 GPUs available in our system. The scalability can be increased by using more GPUs.

We also experiment with different strategies of TensorFlow Distributed Training. Mirrored Strategy works better than the Multi Worker Mirrored Strategy as shown in Table 5.14. The reason might be that
Table 5.12: Computational Result on Different Sparse DNNs using MATLAB on CPU (Serial Implementation)

<table>
<thead>
<tr>
<th>Neurons per Layer</th>
<th>Layer</th>
<th>Connection (Edges) ×10^6</th>
<th>Time (sec)</th>
<th>Rate ×10^9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>120</td>
<td>3.93</td>
<td>126.3</td>
<td>1.87</td>
</tr>
<tr>
<td></td>
<td>480</td>
<td>15.7</td>
<td>466.7</td>
<td>2.02</td>
</tr>
<tr>
<td></td>
<td>1920</td>
<td>62.9</td>
<td>1862.01</td>
<td>2.03</td>
</tr>
<tr>
<td>4096</td>
<td>120</td>
<td>15.7</td>
<td>921.2</td>
<td>1.02</td>
</tr>
<tr>
<td></td>
<td>480</td>
<td>62.9</td>
<td>3540.7</td>
<td>1.07</td>
</tr>
<tr>
<td></td>
<td>1920</td>
<td>252</td>
<td>16579</td>
<td>0.91</td>
</tr>
<tr>
<td>16384</td>
<td>120</td>
<td>62.9</td>
<td>3990.3</td>
<td>0.95</td>
</tr>
<tr>
<td></td>
<td>480</td>
<td>252</td>
<td>21626</td>
<td>0.7</td>
</tr>
</tbody>
</table>

Table 5.13: Computational Result on Different Sparse DNNs using Python TensorFlow on both CPUs and GPUs, Speedup calculated w.r.t. Baseline MATLAB Serial Implementation [*=Out of GPU Local Memory]

<table>
<thead>
<tr>
<th>Neurons per Layer</th>
<th>Layer</th>
<th>Connection (Edges) ×10^6</th>
<th>Rate ×10^9 /sec</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>CPU</td>
<td>1 GPU 2 GPU</td>
</tr>
<tr>
<td>1024</td>
<td>120</td>
<td>3.93</td>
<td>3.23</td>
<td>5.28</td>
</tr>
<tr>
<td></td>
<td>480</td>
<td>15.7</td>
<td>3.37</td>
<td>5.85</td>
</tr>
<tr>
<td></td>
<td>1920</td>
<td>62.9</td>
<td>3.01</td>
<td>5.38</td>
</tr>
<tr>
<td>4096</td>
<td>120</td>
<td>15.7</td>
<td>1.71</td>
<td>2.46</td>
</tr>
<tr>
<td></td>
<td>480</td>
<td>62.9</td>
<td>1.68</td>
<td>2.82</td>
</tr>
<tr>
<td></td>
<td>1920</td>
<td>252</td>
<td>1.43</td>
<td>*</td>
</tr>
<tr>
<td>16384</td>
<td>120</td>
<td>62.9</td>
<td>1.51</td>
<td>2.59</td>
</tr>
<tr>
<td></td>
<td>480</td>
<td>252</td>
<td>1.13</td>
<td>*</td>
</tr>
</tbody>
</table>

Multi Worker Mirrored Strategy has Experimental Support whereas Mirrored Strategy is fully supported in TensorFlow. Mirrored Strategy as well as the Multi Worker Mirrored Strategy support different cross device communication methods. We experiment with each communication schemes and find out the best one. In case of Mirrored Strategy, the default scheme Nccl All Reduce outperforms the other two and takes less time shown in Figure 5.11. While experimenting with Multi Worker Mirrored Strategy, we find that Nvidia’s NCCL works faster compared to the ring-based collectives using gRPC as the communication layer depicted in Figure 5.12.

5.2.7 Conclusion and Future Work

Deep Neural Networks have become one of the prominent research topics in recent times to support modern Artificial Intelligence activities in current world. We face scalability difficulties while working with deeper neural networks falling under the domain of Big Data. The Sparse DNN Challenge given by
Table 5.14: Performance Comparison of Different Strategies for Distributed Training in TensorFlow

<table>
<thead>
<tr>
<th>Neurons per Layer</th>
<th>Layer</th>
<th>Mirrored</th>
<th>Speedup</th>
<th>Multi Worker Mirrored</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,024</td>
<td>120</td>
<td>26.91</td>
<td>4.69</td>
<td>31.81</td>
<td>3.97</td>
</tr>
<tr>
<td></td>
<td>480</td>
<td>118.75</td>
<td>3.93</td>
<td>127.22</td>
<td>3.67</td>
</tr>
<tr>
<td></td>
<td>1,920</td>
<td>410.99</td>
<td>4.53</td>
<td>470.06</td>
<td>3.96</td>
</tr>
<tr>
<td>4,096</td>
<td>120</td>
<td>227.53</td>
<td>4.05</td>
<td>249.81</td>
<td>3.69</td>
</tr>
<tr>
<td></td>
<td>480</td>
<td>785.05</td>
<td>4.51</td>
<td>867.33</td>
<td>4.08</td>
</tr>
<tr>
<td>16,384</td>
<td>120</td>
<td>957.56</td>
<td>4.17</td>
<td>1,015.23</td>
<td>3.93</td>
</tr>
</tbody>
</table>

MIT/IEEE/AmazonGraphChallenge.org focuses on developing new solutions to help the community solving the problems in the domain of graph analytics, machine learning, sparse dataset, big data, high performance computing, and visual analytics. In our work, we provide a solution to the Sparse DNN Challenge using Python TensorFlow. We have achieved $4.7\times$ speedup in data parallel mode in GPU over the baseline sequential MATLAB implementation. We have compared different strategies of TensorFlow Distributed Training and showed their performance. Our work will help the scientific community who use TensorFlow for deep learning in their application to achieve high performance. We get $2.9\times$ speedup using only 2 GPUs over our CPU implementation. We will perform experiments on large number of GPUs in other systems to show the scalability of our solution in our future work. In future, we plan to compare our performance with GraphBLAS implementation and other approaches in the same system used by those work. We will also work towards model parallelism and hybrid parallel implementations for Sparse DNNs.
Chapter 6

Big Graph Machine Learning Applications

6.1 Detecting Web Spam in Webgraphs with Predictive Model Analysis

Web spam is a serious threat for both end-users and search engines (w.r.t., query cost). Webgraphs can be exploited in detecting spam. In the past, several graph mining techniques were applied to measure metrics for pages and hyperlinks. In this study, we justify the importance of webgraph to distinguish spam websites from non-spam ones based on several graph metrics computed for a labelled dataset (WEBSPAM-UK2007) and justify our model by testing on uk-2014 dataset, the most recently available dataset on the same (uk) domain. WEBSPAM-UK2007 dataset includes 0.1 million different hosts and four kinds of feature sets: Obvious, Link, Transformed Link and Content. We use five prominent machine learning (ML) techniques (i.e., Support Vector Machine (SVM), K-Nearest Neighbor (KNN), Logistic Regression, Naive Bayes and Random Forest) to build a ML-based classifier. To evaluate the performance of our classifier, we compute accuracy and F-1 score and perform 10-fold cross validation. We also compare graph based features with content based textual features and find that graph properties are similar or better than text properties. We achieve above 99% training accuracy for most of our machine learning models. We test our model with uk-2014 dataset with 4.7 million hosts for the graph-based feature sets and achieve accuracy in between 90-94% for most of the models. To the best of our knowledge, prior works on web spam detection with WEBSPAM-UK2007 dataset did not use different test dataset for their models. Our model classifier is
capable of detecting web spam for any input webgraph based on its graph metrics features.

6.1.1 Introduction

The study of webgraphs has a significant importance in Web mining, i.e., learning useful structural and organizational information of the Web [197, 198, 199]. Webgraph is a graph having static HTML pages as nodes (vertices) and directed hyperlinks among the pages as edges [61]. In graph (network) mining, computing various structural properties of webgraphs is challenging due to the size of such graphs. Webgraph has great research potentials concerning web security.

Detecting web spam is one of the aspects among several security vulnerabilities. Web spam is a technique being used by some websites to appear in search engines with high rank but low quality. Cloaking, link spam, buying backlinks, content spam, URL spam, redirection, etc., are some of the tactics of web spamming. Link spam consists of the creation of a link structure, usually a tightly knit community of links, aimed at affecting the outcome of a link-based ranking algorithm. Content spam is done by maliciously crafting the content of web pages [200], for instance, by inserting keywords that are more related to popular query terms than to the actual content of the pages. Cloaking consists of sending different content to a search engine than to the regular visitors of a web site [201]. The aforementioned disastrous effects of web spam motivates our work. Archives [202] are becoming more and more concerned about spam in view of the fact that, under different measurement and estimates, roughly 10% of the websites and 20% of the individual pages constitute spam. The above figures directly translate to 10% to 20% waste of archive resources in storage, processing and bandwidth with a permanent increase. The increasing resource waste will question the economic sustainability of the preservation effort in the near future [203].

Webgraph is a potential source of detecting web spam based on graph based features. Emerging graph mining techniques can be used to detect spam in a scalable manner considering the large size of webgraph. Triangle count, clustering coefficient, triangular density, vertex jaccard similarity, vertex cosine similarity, and centrality measures are among potential features of either pages or hyperlinks to be used as features calculated from webgraphs rather than the contents of the pages. During our study, we face the difficulty to find labelled data of spam/non-spam. It shows the need for a machine learning classifier to predict spam based on the currently available labelled dataset. For this reason, we choose the classic WEBSPAM-UK2007 labelled dataset for developing our model classifier.

Web spam filtering, the domain of devising methods to detect useless and spam web content with the target of manipulating search engine results, has drawn much attention in the past years[204, 205, 206]. Recently the achievements against the ‘classical’ web spam seems to be in slow pace [207] and the focus of
researchers has apparently altered towards closely related areas such as spam in social networks [208, 209, 210]. In this study, we emphasize on detecting web spam from webgraph. We figure out the best machine learning technique for each feature set. We compare how the performance vary between graph based features and text based features. We generate graph-based feature set from webgraph for our test dataset and can be applied to any webgraph for feature generation. Our model is tested on different dataset and achieve around 94% accuracy. We also analyze if there is any performance change using different machine learning tools, e.g., Scikit-learn [211] and Weka [212].

The rest of this paper is organized as follows. We describe the related work in Section 6.1.2. Background of machine learning techniques are discussed in Section 6.1.3. In Section 6.1.4, we describe our dataset, our machine learning classifiers, our validation approaches, generating features for testing data, and improvements to the models with feature selection. A detailed analysis of the results is described in Section 6.1.5. Finally, Section 6.1.6 summarizes the findings and concludes the paper with a discussion of future possibilities.

### 6.1.2 Related Works

Several works have been done on web spam but only few focus on webgraph’s graph properties to detect web spam. Most of the works have been done based on link spam, content spam and cloaking. Erdélyi et al. investigated how much various classes of web spam features, some requiring very high computational effort, added to the classification accuracy [213]. Zhou et al. developed novel and effective detection methods for link spam target pages using page farms [214]. Ntoulas et al. devised methods for detecting content spam using classifiers [215]. Mishne et al. used language model disagreement [216] for link spam detection. These content spam detection techniques were similar to spam in e-mail. Chellapilla et al. [201] proposed estimating query popularity and monetizability by analyzing search engine query logs and online advertising click-through logs, respectively. They also presented a new measure for detecting cloaked URLs that used a normalized term frequency ratio between multiple downloaded copies of web pages.

Becchetti et al. used triangle count and clustering coefficient to show the distinction between spam and non-spam [197]. Becchetti et al. later extended their work with several link and node based graph metrics for web spam detection [217]. Castillo et al. presented a spam detection system that used the topology of the webgraph by exploiting the link dependencies among the web pages as well as the content of the pages [218]. Authors in [64] used TWSVM (Twin SVM) with two non-linear kernels for spam page detection using WEBSPAM-UK2007, the same dataset as ours. Again, many works [62, 63, 64] have been done using WEBSPAM-UK2007 for spam detection recently but different from ours. Their work did not emphasize on
the graph-based features generated from the webgraph. Iqbal et. al [63] showed that Random Forest is the best classifier for the given dataset. But they have not provided the detailed implementation (i.e. values for the parameters of the machine learning models used) for their work. Therefore, we can not compare our work with theirs. In our experimentation, we have found SVM reigning over the other models during training. The main reason of this distinction might be that they only focused on detecting spam on any of the features irrespective of the feature type. They intermingled all of the feature sets and found Random Forest working best for their model. But we mainly focused on revolving our result through Link based and Transformed Link based features those can be generated from the webgraphs, along with we also analyzed Obvious and Content feature sets as well to gain insights. Some other works [65, 66] also contributed to detect web spam using different machine learning models and datasets. Nevertheless none of those used a different test dataset to show the test accuracy of the models. Besides, they used pre-computed feature sets for the labelled dataset and did not provide any idea how to generate features for different data to test their models. They did not test their model on a different dataset and reported training accuracy only. We have tested our model on a different most recently available dataset, uk-2014 on the same uk domain, and achieved at most 94% accuracy.

6.1.3 Preliminaries

In this section we discuss the basics of machine learning techniques we have used throughout the paper.

6.1.3.1 Machine Learning Models

Here we describe the well-known machine learning algorithms for classification that we have used in our experiments.

6.1.3.1.1 Support Vector Machine  Support Vector Machine (SVM) is a supervised machine learning algorithm which can be used for both classification or regression challenges. However, it is mostly used in classification problems [219]. SVM works by finding a line that best separates the data into two groups. This is done using an optimization process that only considers those data instances in the training dataset that are closest to the line that best separates the classes. The instances are called support vectors, hence the name of the technique [220]. Different SVM algorithms use different types of kernel functions. These functions can be of different types. For example, linear, nonlinear, polynomial, radial basis function (RBF), and sigmoid. The kernel functions return the inner product between two points in a suitable feature space. A linear kernel is used as normal dot product between any two given observations. A polynomial kernel is a
more generalized form of the linear kernel. The polynomial kernel can distinguish curved or nonlinear input space. RBF can map an input space in infinite dimensional space. Equations for some of the SVM kernels are given in Table 6.1 [221, 222, 223].

Table 6.1: Kernel Functions of SVM

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Equation</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>$K(x, x_i) = \text{sum}(x_1 x_i)$</td>
<td></td>
</tr>
<tr>
<td>Polynomial</td>
<td>$K(x, x_i) = 1 + \text{sum}(x_1 x_i)^d$</td>
<td>$d$ is the degree of the polynomial</td>
</tr>
<tr>
<td>RBF</td>
<td>$K(x, x_i) = \exp(-\gamma \text{sum}((x - x_i^2)))$</td>
<td>$\gamma = [0, 1]$</td>
</tr>
<tr>
<td>Sigmoid</td>
<td>$K(x, x_i) = \tanh(\alpha x^T y + c)$</td>
<td>$\alpha$ is the slope</td>
</tr>
</tbody>
</table>

SVM does not perform well when the dataset is very large— that is due to the required training time becoming higher. It also does not perform very well if the data set has significant noise, i.e., the target classes are overlapping [224]. Considering our dataset size as well as classes, we choose SVM for our classifier.

6.1.3.1.2 K-Nearest Neighbor The K-Nearest-Neighbor (KNN) is a non-parametric classification method. It is a simple algorithm that stores all available cases and classifies new cases based on a similarity measure (e.g., distance functions given in Table 6.2). One of the most popular choices to measure this distance is known as Euclidean. KNN classifier requires storing the whole training set and may be too costly when this set is large [225]. KNN algorithm also supports both classification and regression [226]. An object is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its $K$ nearest neighbors ($K$ is a positive integer, typically small). If $K = 1$, then the object is simply assigned to the class of that single nearest neighbor.

Table 6.2: Distance Functions for KNN

<table>
<thead>
<tr>
<th>Distance Function</th>
<th>Equation</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Euclidean</td>
<td>$D(x, p) = \sqrt{(x - p)^2}$</td>
<td>$x$ and $p$ are the query point and a case from the examples sample</td>
</tr>
<tr>
<td>Euclidean squared</td>
<td>$D(x, p) = (x - p)^2$</td>
<td></td>
</tr>
<tr>
<td>City-block</td>
<td>$D(x, p) =</td>
<td>(x - p)</td>
</tr>
<tr>
<td>Chebyshev</td>
<td>$D(x, p) = \text{Max}(</td>
<td>(x - p)</td>
</tr>
</tbody>
</table>

6.1.3.1.3 Logistic Regression Logistic regression is a binary classification algorithm [227]. The algorithm learns a coefficient for each input value, which are linearly combined into a regression function and transformed using a logistic (s-shaped) function shown in Equation 6.1. The function maps any real value into another value between 0 and 1. Logistic regression is a fast and simple technique, but can be very
effective on some problems [220].

\[ S(z) = \frac{1}{1 + e^{-z}} \]  

(6.1)

where,

\( s(z) = \) output between 0 and 1 (probability estimate)

\( z = \) input to the function (algorithm's prediction e.g. \( mx + b \))

\( e = \) base of natural log

6.1.3.1.4 Naïve Bayes Naïve Bayes uses a simple implementation of Bayes Theorem (hence naïve) where the prior probability for each class is calculated from the training data and assumed to be independent of each other (technically called conditionally independent). There are multiple variations of the Naïve Bayes algorithm depending on the distribution of \( P(x_i \mid y) \). Three of the commonly used variations are Gaussian, Multinomial and Bernoulli [228]. The Gaussian Naïve Bayes algorithm assumes distribution of features to be Gaussian or normal, i.e.,

\[ P(x_i \mid y) = \frac{1}{\sqrt{2\pi \sigma^2_y}} \exp\left( -\frac{(x_i - \mu_y)^2}{2\sigma^2_y} \right) \]

where, \( P(x_i \mid y) \) denotes the conditional probability of an object with a feature vector \( x_i \) belonging to a particular class \( y \)

\( \sigma = \) standard deviation, \( \mu = \) mean

The Multinomial Naïve Bayes algorithm is used when the data is distributed multinomially, i.e., multiple occurrences matter a lot. The Bernoulli algorithm is used when the features in the data set are binary-valued. The decision rule for Bernoulli Naïve Bayes is based on Equation 6.2. It explicitly penalizes the non-occurrence of a feature \( i \) that is an indicator for class \( y \) [211].

\[ P(x_i \mid y) = P(i \mid y)x_i + (1 - P(i \mid y))(1 - x_i) \]  

(6.2)

Naïve Bayes has been shown to be a very effective classification algorithm [220]. The Naïve Bayes classifier is surprisingly effective in practice since its classification decision may often be correct even if its probability estimates are inaccurate [229].

6.1.3.1.5 Random Forest Random forests are used for robust classification, regression and feature selection analysis. Random Forests are an ensemble of \( k \) untrained Decision Trees (trees with only a root node) with \( M \) bootstrap samples (\( k \) and \( M \) do not have to be the same) trained using a variant of the random subspace method or feature bagging method [230]. It is very user-friendly in the sense that it has
only two parameters (the number of variables in the random subset at each node and the number of trees in
the forest), and is usually not very sensitive to their values [231].

6.1.4 Methodology

We describe our dataset, the building of our machine learning classifier, feature selection and validation in
this section.

6.1.4.1 Dataset

We have worked with publicly available WEBSPAM-UK2007 dataset [232] consisting of
105,896,555 nodes representing pages and approximately 3.7 billion edges representing hyperlinks to train
our model. The collection contains 114,529 different hosts. The dataset was collected by the research group
of the Laboratory of Web Algorithmics at the Università degli Studi di Milano. Within the labelled dataset
5.19% was labelled as ‘spam’ and 88.33% was ‘non-spam’. The rest 6.48% was labelled ‘undecided’. We
have not included the undecided data in our classification and filtered out ‘spam’ and ‘non-spam’ within the
labelled dataset. We have used pre-computed feature set calculated from webgraph and html contents of
the pages. The features are computed from the full webgraph for graph-based features. So eliminating the
undecided points for building the model does not affect the already computed feature values. The reason is,
the features reflect the connectivity of the network, different network properties irrespective of a particular
host is taking part in classification model or not. Connectivity among particular groups (spam/non-spam)
or the intra-connection among a particular group is not being taken into consideration in current feature
set. So, omitting the undecided points does not impact the values of feature sets. A brief description of
the features are described in Table 6.3. The obvious feature set has 2 features: the number of pages in
the host and the number of characters in the host name. The Link Feature set has been computed from
the following graph metrics: PageRank, in-degree, out-degree, Truncated PageRank, and TrustRank. The
detailed description of some of the features are given in Subsection 6.1.4.2. The Link Feature set consists
of features taking the logarithms and ratio of the features from Link Feature Set. Content Feature Set has
features generated from text by counting words in the webpages.

Although the data was crawled a long time back, we have chosen to work with it for some specific reasons.
The unavailability of labelled data is one of the main reasons. Another important reason is that we mainly
want to focus on the webgraph properties and those specific values related to spam class. Also, our work is
comparable to others who have used the same dataset very recently. As, we do not emphasize on the text
based classification of spam that changes over time, it is quite reasonable to work with a well-known labelled
dataset.

Table 6.3: Feature Sets used in our Experimentation

<table>
<thead>
<tr>
<th>Feature Set</th>
<th>Source</th>
<th>Feature Description</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obvious Graph</td>
<td>Graph</td>
<td>the number of pages in the host and the number of characters in the host name</td>
<td>2</td>
</tr>
<tr>
<td>Link Graph</td>
<td>Graph</td>
<td>in-degree, out-degree, pagerank and more [217]</td>
<td>41</td>
</tr>
<tr>
<td>Transformed Link</td>
<td>Graph</td>
<td>ratio of indegree and outdegree, average, reciprocite, log and more [217]</td>
<td>137</td>
</tr>
<tr>
<td>Content Text</td>
<td>Text</td>
<td>number of words in the home page, average word length, average length of the title, etc., for a sample of pages on each host [218]</td>
<td>96</td>
</tr>
</tbody>
</table>

For Testing our validated model classifier, we have worked with a different dataset that is uk-2014 webgraph [233, 234, 235]. This graph is a large snapshot of the .uk domain taken at the end of 2014. The maximum number of pages per host was set to 10000. The webgraph has 787.8 million nodes and 107 billion edges. The total number of hosts is 4.7 millions, the number of instances for our test dataset.

6.1.4.2 Feature Generation for Test Data

Figure 6.1: Overview of Feature Generation Steps

We have computed the features for Link and Transformed Link Feature Sets by computing the graph metrics from the uk-2014 webgraph. The steps to generate the feature sets is given in Fig. 6.1. We have included only those features where the computation involves PageRank, in-degree and out-degree. We have omitted features involving computation of TrustRank and TruncatedPageRank because the computation for these metrics for the labelled dataset is not described and may introduce variation in values. For each host, we include the computation for both home page and the page with maximum PageRank.
Algorithm 18: Graph-based Feature Generation from Webgraph

Data: Input Webgraph, \( G \) in BVGraph Format; Host Graph, \( G_H \)
Result: Feature Sets

for Each node in \( G \) do
    calculate_Pagerank()
    calculate_In-Degree()
    calculate_Out-Degree()
end

for Each node in \( G \) do
    match_url(\( G, G_H \))
    set_pair(node-id, host-id)
end

for Each node \( i \) in \( G_H \) do
    host_list ← get_list(i)
    find_homepage(i)
    for Each node \( j \) in host_list do
        \( \text{PageRank\_Max}[i] ← \text{MAX}(\text{PageRank}(j)) \)
        \( \text{node\_max} ← j \)
        /* Calculate features for Link Feature Set */
        in_degree\_i ← SUM(In-degree(j))
        out_degree\_i ← SUM(Out-degree(j))
        PageRank\_i ← AVG(PageRank(j))
        /* ............... Calculate Rest Features ............... */
    end
end

/* Calculate features for Transformed Link Feature Set from the previous generated features */

6.1.4.2.1 Link Feature Set The features we compute for Link Feature Set are the following: \text{In-degree}, \text{Out-degree}, \text{PageRank}, \text{Assortativity coefficient} (ratio of degree and average degree of neighbors), Average \text{in-degree of out-neighbors}, Average \text{out-degree of in-neighbors}, Standard deviation of the PageRank of in-neighbors. So, in total we get 15 features for Link Feature Set.

6.1.4.2.2 Transformed Link Feature Set In case of Transformed Feature Set, we similarly compute for both home page and page with maximum PageRank. We take logarithm of each of the previous features for the Transformed Feature Link Set. Besides, we also compute \text{Ratio of Log of sum of the in-degree of out-neighbors, Log of sum of the out-degree of in-neighbors, Log of ratio of in-degree and PageRank, Log of ratio of out-degree and PageRank, log of ratio of Standard deviation of the PageRank of in-neighbors and PageRank}. Total number of features for Transformed Link Feature Set is 25.

Pseudocode for feature generation is given in Algorithm 18
6.1.4.3 Environment

The experiments have been performed on an Intel Core i7-4770 CPU @ 3.4GHz×8 processor and 16 GB RAM machine. During feature generation, we have used MATLAB to compute the in-degree, out-degree, PageRank from uk-2014 webgraph. Further we use Python Pandas DataFrame to calculate the features. We have used Python Scikit-learn [211] [236] and Weka (Java) [212] [237] to build our classifier based on various machine learning models.

6.1.4.4 Building Classifier

We have used 5 well-known machine learning models to build our classifier. In machine learning there are no best algorithms according to Wolpert’s “no free lunch’ theorem. Some algorithms work better with some application or data. So we have chosen some of the best algorithms that might work better for our dataset. A brief description of tweaking the parameters for each of the models to build our classifier is described in this section.

6.1.4.4.1 SVM  For Scikit-learn, we have used different kernels (linear, polynomial, sigmoid, rbf) and gamma values. We get the best result using kernel=rbf, C=1 and gamma=scale. Whereas for Weka, the best accuracy is achieved with polynomial kernel.

6.1.4.4.2 KNN  At first, we have determined the optimal value of $K$ from Scikit-learn shown in Fig. 6.2. We have used the same value of $K$ in Weka as well. The optimal values of $K$ for Obvious Feature Set are 6 and the values greater than 7. The maximum accuracy for Link Feature Set can be found for $k = 8, 10; k > = 20$. For Transformed Link Feature Set, $K = 10; K >= 20$ are the optimal values of $K$. The optimal value for Content Feature Set is 4.

6.1.4.4.3 Logistic Regression  We have used Scikit-learn default parameters with saga solver and 50000 maximum iterations to get the best accuracy. Weka in its default setting shows the best accuracy.

6.1.4.4.4 Naïve Bayes  We have used both Gaussian and Multinomial Naïve Bayes for Scikit-learn as well as Weka, but better result has been found from Gaussian Naïve Bayes.

6.1.4.4.5 Random Forest  In Scikit-learn as well as Weka we have used $max\_features = \text{sqrt}(n\_features), depth = 0, seed = 1$ and $max\_iterations = 100$. Other parameters have been kept default for both Scikit-learn and Weka.
6.1.4.5 Validation

To validate our model, we have chosen k-fold Cross Validation. k-fold Cross Validation ensures the full dataset is used to train the model. We have used widely accepted value $k = 10$.

We also measure the F1 score of our model. F1 score is harmonic mean of precision and recall. Having a higher Recall means there are less FALSE NEGATIVES. As much as less False Negatives or Zero FN means, model prediction is really good. Whereas having higher Precision means, there are less FALSE POSITIVES. Similarly, Less or Zero False Positives means Model prediction is really good. Thus having a higher F1 score implies good Model Prediction.

6.1.4.6 Feature Selection

We have used 'Select Attributes' functionality of Weka for selecting features to improve our classifier. We have used Information Gain as well as Gain Ratio functions to select the attributes. We eliminate the attributes with 0 value for both functions. Both functions choose the same attributes for elimination.
6.1.4.7 Testing Classifier with uk-2014 dataset

Our Test data uk-2014 has 4.7 million instances. Weka runs out of memory if we try to load the full dataset at once. So we divide the test data into multiple chunks for testing. Finally, we take the average of accuracy and F-Measure found from all of the chunks to get result for the full dataset.

6.1.5 Result

In this section, we describe several analyses with the derived results for evaluating the accuracy of our classification model.

Table 6.4: 10-fold Cross Validation Accuracy for Scikit-learn and Weka

<table>
<thead>
<tr>
<th>Model</th>
<th>Obvious Scikit-learn</th>
<th>Obvious Weka</th>
<th>Link Scikit-learn</th>
<th>Link Weka</th>
<th>Transformed Link Scikit-learn</th>
<th>Transformed Link Weka</th>
<th>Content Scikit-learn</th>
<th>Content Weka</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM</td>
<td>94.44</td>
<td>100</td>
<td>94.42</td>
<td>100</td>
<td>100</td>
<td>99.95</td>
<td>94.60</td>
<td>99.97</td>
</tr>
<tr>
<td>KNN</td>
<td>94.45</td>
<td>100</td>
<td>94.45</td>
<td>99.52</td>
<td>94.45</td>
<td>96.27</td>
<td>94.91</td>
<td>99.82</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>94.45</td>
<td>99.95</td>
<td>94.1</td>
<td>99.8</td>
<td>93.75</td>
<td>99.67</td>
<td>94.54</td>
<td>99.71</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>92.87</td>
<td>99.2</td>
<td>73.11</td>
<td>98.2</td>
<td>83.44</td>
<td>94.17</td>
<td>6.99</td>
<td>36.01</td>
</tr>
<tr>
<td>Random Forest</td>
<td>90.75</td>
<td>100</td>
<td>99.98</td>
<td>100</td>
<td>99.45</td>
<td>96.97</td>
<td>99.4</td>
<td>98.47</td>
</tr>
</tbody>
</table>

Table 6.5: F-measure for Scikit-learn and Weka

<table>
<thead>
<tr>
<th>Model</th>
<th>Obvious Scikit-learn</th>
<th>Obvious Weka</th>
<th>Link Scikit-learn</th>
<th>Link Weka</th>
<th>Transformed Link Scikit-learn</th>
<th>Transformed Link Weka</th>
<th>Content Scikit-learn</th>
<th>Content Weka</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM</td>
<td>0.9</td>
<td>1</td>
<td>0.92</td>
<td>1</td>
<td>1</td>
<td>0.999</td>
<td>0.92</td>
<td>1</td>
</tr>
<tr>
<td>KNN</td>
<td>0.92</td>
<td>1</td>
<td>0.92</td>
<td>0.995</td>
<td>0.92</td>
<td>0.954</td>
<td>0.93</td>
<td>0.997</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>0.92</td>
<td>0.999</td>
<td>0.93</td>
<td>0.998</td>
<td>0.92</td>
<td>0.997</td>
<td>0.92</td>
<td>0.997</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>0.92</td>
<td>0.992</td>
<td>0.83</td>
<td>0.983</td>
<td>0.85</td>
<td>0.95</td>
<td>0.03</td>
<td>0.471</td>
</tr>
<tr>
<td>Random Forest</td>
<td>0.95</td>
<td>1</td>
<td>0.97</td>
<td>1</td>
<td>0.99</td>
<td>0.964</td>
<td>0.98</td>
<td>0.984</td>
</tr>
</tbody>
</table>

6.1.5.1 10-fold Cross Validation Accuracy

Table 6.4 represents the 10-fold cross validation accuracy for Python Scikit-learn and Java Weka for each of the models and feature sets. For Scikit-learn, we can see that SVM has 100% accuracy for Transformed Link
feature Set and 94% accuracy for the rest feature sets. KNN and Logistic Regression both have around 94% accuracy for all of the feature sets. For Naïve Bayes only Obvious feature set shows a better accuracy of 92.87%. All feature sets show accuracy above 99.4% except Obvious feature set for Random Forest Model.

Again, for Weka, we find that SVM has 99.9-100% accuracy for all of the feature sets. KNN has above 99.5% accuracy for all feature sets except Transformed Link feature set. For all feature sets, Logistic Regression Model has accuracy above 99.67%. Content Feature Set does not fit to Naïve Bayes Model having a poor accuracy whereas Obvious and Link feature sets show accuracy greater than 98.2%. Obvious and Link feature sets show 100% accuracy for Random Forest Model. Overall, Obvious and Link feature sets show better accuracy of 99.6% on average for all of the models compared to the other two feature sets.

6.1.5.2 F-measure

We have calculated the F-measure score of all of the feature sets for each of the machine learning models shown in Table 6.5. Most of the values are higher and closer to 1 indicating high precision and recall of our models. In our classification task, we intend to build a classifier with high precision and recall. Our Model decides a website is spam or non-spam. We want our model to do the following:

- precisely identify non-spam websites from spam websites (precision)
- identify each website from both spam and nos-spam classes (recall)

It means that we need to select the model that performs well on both metric. So, a high F1 score indicates such model.

By comparing Table 6.4 and Table 6.5, we see that the accuracy and F-measure values do not deviate at all that indicates the uneven class distribution does not affect our model. It implies accuracy is a good measure.

6.1.5.3 Finding Best Model for each Feature Set

In this section we have compared accuracy for all machine learning models and found the best Machine Learning Model for each feature set. For brevity, we have not included the values of F-measure in Table 6.6 as both accuracy and F-measure indicate similar result in our model.

6.1.5.3.1 Obvious Feature Set  SVM, KNN and Random Forest Models show 100% accuracy in Weka. Again, Naïve Bayes and Logistic Regression Models also have accuracy above 99% in Weka. So, all of the Machine Learning Models are well-fitted for Obvious Feature Set.
Table 6.6: Machine Learning Model and Tool Selection for each Feature Set

<table>
<thead>
<tr>
<th>Feature Sets</th>
<th>SVM</th>
<th>KNN</th>
<th>L.R.</th>
<th>N.B.</th>
<th>R.F.</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>94.44, 100</td>
<td>1</td>
<td>94.45, 100</td>
<td>1</td>
<td>94.45, 99.95</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>94.42, 100</td>
<td>2</td>
<td>94.45, 99.52</td>
<td>4</td>
<td>94.1, 99.8</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>100, 99.95</td>
<td>1</td>
<td>94.45, 96.27</td>
<td>4</td>
<td>93.75, 99.67</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>94.6, 99.97</td>
<td>2</td>
<td>94.91, 99.82</td>
<td>3</td>
<td>94.54, 99.71</td>
<td>4</td>
</tr>
</tbody>
</table>

Here, A.=Accuracy(%), R.=Rank, S.=Scikit-learn, W.=Weka

6.1.5.3.2 Link Feature Set  Random Forest Model shows around 100% accuracy in Weka as well as Scikit-learn. SVM also shows 100% accuracy in Weka. Again, KNN, Naïve Bayes and Logistic Regression Models show above 98% accuracy in Weka. So, all of the Machine Learning Models are well-fitted for Link Feature Set.

6.1.5.3.3 Transformed Link Feature Set  We get around 100% accuracy for SVM shown in both Scikit-learn and Weka. Above 99% accuracy is found for Random Forest Model in Scikit-Learn and for Logistic Regression in Weka. KNN and Naïve Bayes perform moderate. So, SVM is best-fitted for Transformed Link Feature Set considering both Scikit-learn and Weka.

6.1.5.3.4 Content Feature Set  For Content Feature Set, SVM, KNN and Logistic Regression Models show around 100% accuracy in Weka. Random Forest Model has above 98% accuracy in both Scikit-learn and Weka but Naïve Bayes Model performs poorly and cannot be considered for Content Feature Set.

We have summarized the results in Table 6.6. We have ranked the machine learning models in ascending order according to highest to lowest accuracy for each feature set. For instance, according to rank, the best
machine learning models for Transformed Link Feature Set are SVM, Random Forest, Logistic Regression, KNN and Naïve Bayes respectively. We have also determined the best modelling tool for each of the feature sets. To illustrate, KNN, Logistic Regression and Naïve Bayes Models are well-suited to Weka only whereas Random Forest Model is well-suited to Scikit-learn but SVM works well for both Weka and Scikit-learn for Transformed Link Feature Set. We can see that except Random Forest Model for Transformed Link Feature Set, Weka gives better performance for all of the models and feature sets. For Scikit-learn only Random Forest Model performs well for all the feature sets except Obvious Feature Set.

Regarding the machine learning models, SVM is faster in training, better in accuracy with stability / robustness and works well for each of the Feature Sets. Random Forest is good for balancing error in class population unbalanced data sets reflected in our case. Logistic regression assumes no error in the output variable (y). As we consider removing outliers and possibly misclassified instances from our training data, it works well and is reflected in our result. KNN is a very simple and easy algorithm that even works well for our dataset. Naïve Bayes performs well in case of categorical input variables compared to numerical variables. As our feature sets consist mostly of numerical values, it works poorly for all feature sets specifically, Content Feature Set.

6.1.5.4 Feature Selection for Improvement in Naïve Bayes Model

The accuracy of Naïve Bayes Model with Scikit-learn is less than 90% for all of the feature sets. So, we remove some of the attributes in order to improve the accuracy. Based on the criteria as explained in Section 6.1.4.6, we have eliminated several features from our Feature Sets. A summary of our feature selection has been shown in Table 6.7.

<table>
<thead>
<tr>
<th>Feature Set</th>
<th>Total Features</th>
<th>Eliminated Features</th>
<th>Existing Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link</td>
<td>41</td>
<td>7</td>
<td>34</td>
</tr>
<tr>
<td>Transformed Link</td>
<td>137</td>
<td>52</td>
<td>85</td>
</tr>
<tr>
<td>Content</td>
<td>96</td>
<td>14</td>
<td>82</td>
</tr>
</tbody>
</table>

Eliminated Link features include:

- Neighbors at distance 3 of home page
- Fraction of out-links that are also in-links of home page
- Fraction of out-links that are also in-links of page with maximum pagerank
• Assortativity coefficient of the home page (ratio of degree and average degree of neighbors)

• trustrank of home page

• trustrank of page with maximum pagerank

Some of the eliminated features from Transformed Link Set are: Ratio of pagerank of the page with maximum pagerank, pagerank of home page; Ratio of neighbors at distance 4 of home page, neighbors at
Table 6.8: 10-fold Cross Validation Accuracy and F-measure with Link Feature Set and Transformed Link Feature Set for uk-2014 Test Dataset using Scikit-learn and Weka

<table>
<thead>
<tr>
<th>Model</th>
<th>Scikit-learn</th>
<th>Weka</th>
<th>Transformed Link</th>
<th>Weka</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Link</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Accuracy</td>
<td>F Measure</td>
<td>Accuracy</td>
<td>F Measure</td>
</tr>
<tr>
<td>SVM</td>
<td>90.812</td>
<td>0.854</td>
<td>94.5</td>
<td>0.901</td>
</tr>
<tr>
<td>KNN</td>
<td>85.65</td>
<td>0.699</td>
<td>87.5</td>
<td>0.839</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>89.2</td>
<td>0.787</td>
<td>86.67</td>
<td>0.84</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>60.72</td>
<td>0.75</td>
<td>79.4</td>
<td>0.755</td>
</tr>
<tr>
<td>Random Forest</td>
<td>91.25</td>
<td>0.889</td>
<td>92.33</td>
<td>0.865</td>
</tr>
</tbody>
</table>

distance 4 of page with maximum pagerank; Ratio of trustrank of page with maximum pagerank, trustrank of home page and many more. A close observation points out that the features related to trustrank have been eliminated from both Linked and Transformed Link feature sets.

After the selection, we get less than 10% improvement shown in Fig. 6.3. Again, for Link Feature Set the accuracy decreased around 33%. So, selected features did not contribute to an improved performance. For our dataset, feature selection has not been proved to be an acceptable technique.

6.1.5.5 Graph based vs. Text based Feature Sets

Extracting the features from contents of the link/page is somewhat exaggerating process because of huge amount of texts. Whereas calculating several graph-based metrics such as clustering-coefficient, triangle count, ratio of indegree and outdegree etc., from the webgraph is more convenient considering the emerging graph mining techniques. From our analysis we have found that graph metrics based Link Feature Set always provides better or similar accuracy than text based Content Feature Set as shown in Fig. 6.4.

6.1.5.6 uk-2014 Dataset Test Result

Based on our training model classifier, we have tested our model with uk-2014 dataset. We focus only on the Graph based (Link and Transformed Link) Feature Sets. Table 6.8 shows the 10-Fold Cross Validation Accuracy and F-score for these two feature sets using both Scikit-learn and Weka. For Link Feature Set, we get 90-94% accuracy for both Scikit-learn and Weka using SVM and Random Forest models. KNN and Logistic Regression performs moderately with 85-89% accuracy. In case of Transformed Link Feature Set, the accuracy is around 90% for SVM, Logistic Regression and Random Forest model in Weka. SVM and
Random Forest also has 90-91% accuracy with Scikit-learn. Overall, both SVM and Random Forest Model perform well for our test dataset. The higher values of F1-score also indicate good performance of our classifier.

6.1.5.7 Weka versus Scikit-learn

We compare the implementation differences of Weka and Scikit-learn for the Machine Learning Models. While working with both of the modelling tools, we have found some dissimilarities. In both of the tools, some parameters do not match at all. For instance, the parameter setting for Logistic Regression Model for Scikit-learn and Weka are different. Parameters related to Scikit-learn are `penalty`, `dual`, `tol`, `C`, `fit_intercept`, `intercept_scaling`, `class_weight`, `random_state`, `solver`, `max_iter`, `multi_class`, `verbose`, `warm_start`, `n_jobs`, `l1_ratio`. Whereas for Weka the parameters are: `batchsize`, `max_its`, `ridge`, `useConjugateGradient`. Only common parameter between both is maximum iteration.

We could not compare both in the same scale, still we have kept all of the parameters same those match and compared the result. From our analysis in Section 6.1.5.3, we see that Weka works better for all of the models for each of the Feature sets. Exception is for only Random Forest Model for Transformed Link Feature Set perceived from Table 6.6. Overall, we conclude that the default parameter settings for Weka provides better accuracy and is useful for end-users. On the other hand, we need to calibrate the parameters with Trials and Errors to obtain a better accuracy in Scikit-learn. The professional developers are much comfortable with Scikit-learn. Weka saves our valuable time as we need not give much time to adjust the parameters.

6.1.5.8 Comparison with existing work

We face difficulty to compare our work with the existing works as most of the works did not provide the model parameters for reproducibility. Again, the intermingling of feature sets do not match. We then compare the performance of our machine learning classifier with an existing work [64], although some information was missing. The authors developed SVM with two kernels. We have achieved a performance gain in simple SVM with rbf kernel as well as their TWSVM Model for Content Feature Set represented in Fig. 6.5.

We have used same dataset as well as same 10-fold Cross Validation accuracy to compare our result with theirs. The authors have not mentioned the parameter settings of their SVM Model, so we have used our own parameter settings providing the best output and achieved the improved performance.
6.1.6 Conclusion

In this work, we have developed a machine learning classifier to detect web spam from webgraph. For our labelled dataset, WEBSPAM-UK2007, SVM provides 100% accuracy for all of the Feature Sets. Along with Random Forest and KNN are best-suited with 100% accuracy for Obvious Feature Set. For Link Feature Set, Random Forest also provides 100% accuracy. Naïve Bayes shows 94.17%-99.63% accuracy for all of the feature sets except Content Feature Set. We generate features for test data which can be used for any webgraph for feature generation. Our model shows 90-94% accuracy for our test data with the graph-based features generated from uk-2014 dataset, the most recent available webgraph in uk domain. Using our predictive model classifier, we can detect web spam with graph-based features for any webgraph provided as input. We have found that Weka gives better accuracy compared to Scikit-learn for these feature sets in default parameter settings in most cases. In future, we will generate feature sets based on the values of graph metrics other than in-degree, out-degree and PageRank and check how other metrics perform to detect spam and non-spam websites. We plan to compute the graph metrics with Apache-Hadoop-Spark using Graphx because some webgraphs are very large to handle. For scalable computing, we will use the generated feature sets for our classifier built using Apache MLib to predict webspam with high accuracy. We plan to use our classifier in an existing web archive to check if webspam has been archived there and how much space we can save by removing web spam from the archive.
6.2 COVID-19 Vaccination Awareness and Aftermath: Public Sentiment Analysis on Twitter Data and Vaccinated Population Prediction in the USA

Social media, such as Twitter, is a source of exchanging information and opinion on global issues such as COVID-19 pandemic. In this study, we work with a database of around 1.2 million tweets collected across five weeks of April–May 2021 to draw conclusions about public sentiments towards the vaccination outlook when vaccinations become widely available to the population during the COVID-19 pandemic. We deploy natural language processing and sentiment analysis techniques to reveal insights about COVID-19 vaccination awareness among the public. Our results show that people have positive sentiments towards taking COVID-19 vaccines instead of some adverse effects of some of the vaccines. We also analyze people’s attitude towards the safety measures of COVID-19 after receiving the vaccines. Again, the positive sentiment is higher than that of negative in terms of maintaining safety measures against COVID-19 among the vaccinated population. We also project that around 62.44% and 48% of the US population will get at least one dose of vaccine and be fully vaccinated, respectively, by the end of July 2021 according to our forecast model. This study will help to understand public reaction and aid the policymakers to project the vaccination campaign as well as health and safety measures in the ongoing global health crisis.

6.2.1 Introduction

Machine learning (ML) is the most recent method in data science that has paved the way for technological accomplishments and tools that would have been unimaginable a couple of years ago. Image recognition, sentiment analysis [238, 239, 240, 241], product recommendations, spam/fraud detection [82], social media features, etc. are some of the real-world machine learning applications that are sweeping the world. Different web-based social media have been broadly utilized as a means of trading data by both the population and organizations all around the world. The quantity of social media users has started to increment quickly, particularly in the previous decade. Facebook, Twitter, YouTube, LinkedIn, and Pinterest saw huge increments over the previous year. Facebook is the most famous social media with 2.8 billion monthly active users [242], while Twitter has around 300 million monthly active users [243]. Twitter is encountering fast development and is rapidly acquiring fame everywhere in the world. The Twitter interface is utilized by certain users to help different viewpoints, for instance as a medium to fight, political missions, and information spreading, and it is assuming a significant part in social development.
Coronavirus is one of the moving themes on Twitter since January 2020 and has kept on being examined to date. A cluster of pneumonia cases in Wuhan, China, was reported to the World Health Organization (WHO) on 31 December 2019 and the cause of the pneumonia cases (the disease named as COVID-19) was identified as a novel betacoronavirus, the 2019 novel coronavirus (2019-nCoV, renamed as SARS-CoV-2) [244]. In March 2020, COVID-19 was declared as a pandemic by WHO considering more than 118,000 cases in 114 countries [245]. By June 1, 2021, there has been 3.57 million confirmed deaths and 171.19 million confirmed COVID-19 cases [246]. The situation has improved since vaccination of COVID-19 started to scale up. As we gain more evidence of the positive impacts of vaccination on transmission, it will help to strengthen public trust [247]. Considering this, analyzing the public opinion or sentiment is very important for motivating people to be vaccinated against COVID-19.

This paper aims at analyzing public sentiment on COVID-19 vaccination and the aftermath of vaccination regarding health safety measures. We scrape tweets based on different keywords filtering related to vaccines and health and safety issues after vaccination to help us understand public reaction and aid policymakers to project the vaccination campaign as well as health and safety measures. Analyzing the Twitter content empowers health experts, policymakers to learn about the public’s reaction to vaccination during the COVID-19 pandemic. It also elucidates people’s opinions on the health guidelines for the prevention of COVID-19 after getting vaccinated. Discoveries from this analysis related to health are useful as fundamental examinations for building more thorough models, which can be utilized to create proposals for the larger public and establish meaningful strategies and policies. The tweets in this study have caught discussions about vaccination and health guidelines during COVID-19 in numerous nations. Social media information permits scientists and researchers to have a global point of view, which is particularly shrewd during a worldwide pandemic. This study can be replicated by scraping tweets regularly until the COVID-19 pandemic comes to an end for understanding the overall public sentiment while the vaccination campaign is ongoing. In another part of the paper, we try to show the recent forecasting of the US population to have an understanding of the current vaccination scenario in the USA. We aim to have an overview on whether the vaccination campaign is proceeding properly and people are aware of the situation. We find that our forecast model also predicts a similar percentage of population as the US Government is targeting by a certain timeline. If the predicted percentage had been less, it would be an indicator to emphasize more on vaccination campaign. In addition, the health and government officials could plan accordingly beforehand to handle such scenario.

This type of study will be useful for the health and government officials to get insights about any newly discovered disease with early invention of vaccine for that particular disease. The contribution of this paper is two-fold.
• We performed sentiment analysis to have an overview of people’s opinion regarding the COVID-19 vaccination.
  – We analyzed the tweets on seven different types of COVID-19 vaccine and find public sentiments. This analysis is useful to figure out whether people are reluctant to vaccinate considering the side effects of some of the vaccines. In addition, this response indicates people’s interest and showing willingness to take vaccines in response to the vaccination campaign.
  – We collected tweets mentioning vaccination with different terms related to health guidelines. Public sentiment on these tweets help us learn people’s opinion on how they are following the health guidelines after getting vaccinated.
• We performed a time series forecasting on the vaccination scenario of the USA and predict the percentage of population that will be vaccinated by a certain timeline.

6.2.2 Related Works

There have been several works related to analyzing the Twitter dataset on different topics during the COVID-19 pandemic [67, 68, 69, 70]. Only a few studies focus on the Twitter data related to COVID-19 vaccination [71, 72].

Glowacki et al. [67] performed text mining to identify addiction concerns during the COVID-19 pandemic. They captured public tweets containing the two keywords “addiction” and “covid” together and came up with 14 prevalent topics and provided discussion on those topics. Their dataset includes only 3301 tweets. They aim at identifying the public discussion on addiction on Twitter during the COVID pandemic but have not focused on sentiment analysis on addiction due to the pandemic. In [68], the authors worked with Twitter data related to “Mask”. They found that the volume and polarity of mask related tweets has greatly increased during the timeline from March 17, 2020 to July 27, 2020. They also employed clustering techniques to organize these tweets into fifteen high-level themes and fifteen specific topics within each theme. They performed sentiment analysis on each theme and topic. They also applied an abstractive text summarization model using NLP to automatically interpret and describe the subject of the conversation occurring within each theme and topic cluster. Xue et al. [248] used a machine learning approach, Latent Dirichlet Allocation (LDA), to identify popular unigram, bigrams, salient topics and themes, and sentiments in the collected four million tweets on COVID-19 using 25 different hashtags in the period of 1 March 2020 to 21 April 2020. They used the NRC Emotion Lexicon to classify the sentiments into eight primary emotions: anger, anticipation, fear, surprise, sadness, joy, disgust, and trust. Their result show that Twitter users reveal fear when tweeting
about COVID-19 new cases or death rather than trust. Pano and Kashef [69] presented sentiment analysis on tweets related to bitcoins during COVID-19 pandemic using VADER. They compared 13 different text preprocessing strategies for correlating the sentiment scores of the tweets with bitcoin price. [70] performed sentiment analysis using TextBlob on online education by webscraping 154 articles from online news and blogging websites during the COVID-19 pandemic. Their results show that over 90% of the articles are positive, and, in general, the blogs have been more positive than the newspaper articles.

Chen and Dredze [249] were the first to analyze vaccine related images on Twitter. The goal was the identification of propagation of images being used in vaccine-related tweets and to predict with a logistic regression model if the image has been retweeted or not. The authors released the labeled dataset that can be used as sentiment classifier for images. Villavicencio et al. [71] based their study on sentiment analysis of COVID-19 vaccination tweets in the Philippines. The authors used Naïve Bayes model to classify English and Filipino language tweets (993 tweets) using the RapidMiner data science software with 81.77% accuracy and showed the sentiments of Filipinos towards COVID-19 vaccines. Chaudhri et al. [72] recently analyzed whether people are in favor of receiving a COVID-19 vaccine. Their result shows that on average people have weakly positive sentiment in favor of receiving the COVID-19 vaccine shots. However, the authors in their analysis used a very limited number of tweets, only 900 tweets. They did not disclose how they selected those tweets or what things they considered while scraping them. The timeline of scraping the tweets is also not mentioned in the article.

Only the works in [71, 72] are somehow related to our work. However, the work of Villavicencio et al. [71] is related to tweets in the Philippines only, whereas we collect tweets around the globe. For this reason, we have around 1.2 million tweets, but they analyzed only 993 tweets. This work also uses the Naïve Bayes model to predict the classification, whereas we classify the tweets with lexicon-based classifier and use the publicly available tools TextBlob and VADER. Villavicencio et al. [71] performed manual annotation for the training data, i.e., they manually provided the sentiment labels for the training data to predict the test data. We do not predict the sentiment labels, rather we calculate labels using the well accepted sentiment analysis tools. Thus, we cannot show any accuracy comparison with the work of Villavicencio et al. [71]. For the work given in [72], the Twitter data collection criterion and timeline are missing, which are necessary if we want to compare our result with theirs. Their dataset consists of 900 tweets only and is also not publicly available, so we could not compare our work with theirs. In our case, we do a thorough study on people’s sentiment about the COVID-19 vaccines and if they are maintaining a healthy way of life after getting vaccinated. We choose seven different vaccines and collect tweets when vaccines are more accessible to the general public. Our work is reproducible given our implementation details and publicly available code-base. We show how
sentiment analysis can be useful for health and government officials to lead the vaccination campaign based on people’s reaction.

Another contribution of this paper is related to time series forecasting of vaccinated population in the USA. Although there is a rich literature on different time series forecasting methods for many different applications [73, 74, 75], no prior work is relevant to COVID-19 vaccination forecast. The study done by the Centers for Disease Control and Prevention (CDC) [76] predicts the cumulative death for COVID-19, 4 weeks ahead using the ensemble method in R. Using this model, the CDC shows prediction for weekly death/cumulative deaths, daily hospitalization and weekly new COVID-19 cases. However, they still do not integrate any vaccination data for prediction of the vaccination scenario in the USA (state and national). This prediction takes into consideration, e.g., COVID-19 data, demographic data, and mobility data methods and estimates the impacts of interventions (social distancing, use of face coverings, etc.) and not a time series forecast. Another prediction of COVID-19 deaths and cases in the 15 countries of South and Central Europe is done in [77] using ensemble learning of the well known regression methods in WEKA. The dataset and detailed implementation are not well-described, so we cannot check how well their classifier could work on the vaccination dataset. Many other studies are focused on several other predictions: the stock market [250, 75, 251], business sales [252, 253], temperature [254], weather [255], energy consumption [256, 257], electricity [258, 259], etc. We did not find any work that shows a forecast using vaccination data in the USA. Thus, we cannot show a comparison of our model in terms of accuracy with prior works. We do a time series forecasting on the US population to show what percentage of the population will be vaccinated in the near future time. This projection will help to inform public health decisions and policy making by projecting the number in the coming weeks.

6.2.3 Preliminaries

In this section, we describe the different Python libraries used in our work, the evaluation metrics, the methods for sentiment analysis, and the performance metrics for time series forecasting modeling.

6.2.3.1 Sentiment Analysis

There are two major approaches to sentiment analysis.

- Supervised machine learning or deep learning approaches.
- Unsupervised lexicon-based approaches.
As we do not have a pre-labeled dataset, we work with the second approach. In this section, we discuss two well-known sentiment analysis tools, TextBlob [260] and VADER [261].

6.2.3.1.1 TextBlob
TextBlob [260] is a Python library that provides support for different Natural Language Processing (NLP) tasks including sentiment analysis. TextBlob outputs the following two metrics for any input text.

Polarity is a float that lies within $[-1, 1]$. Negative values near $-1$ indicate negative sentiment, positive values near $1$ indicate positive sentiments, and $0$ indicates neutral sentiment.

Subjectivity is also a float which lies in the range of $[0, 1]$. Subjective sentences generally refer to personal opinion, emotions, or judgments, whereas objective refers to factual information. A sentiment that is more objective than subjective receives a lower score, which denotes a more likely-to-be-accurate reading or fact.

6.2.3.1.2 VADER
VADER [261] is a lexicon- and rule-based sentiment analysis tool. It is specifically designed for sentiments expressed on social media and works well on texts from other domains as well.

VADER provides output as the valence scores of each word in the lexicon, as ratios for proportions of text that fall into a positive, negative, or neutral category and all together sums to 1. The compound score is the most commonly used metric for sentiment analysis by researchers. Compound score is a float which lies in the range of $[-1, 1]$. The compound score is computed by summing the valence scores of each word in the lexicon, adjusted according to the rules, and then normalized to be between $-1$ and $+1$. Compound score $\geq 0.05$ indicates positive sentiment and compound score $\leq -0.05$ indicates negative sentiment. Neutral sentiment is defined by $-0.05 < \text{compound score} < 0.05$.

6.2.3.2 Time Series Forecasting

The two general categories of machine learning (ML) are supervised and unsupervised learning. Supervised ML techniques are used when we a datum that we want to predict by using previous data of inputs and outputs to predict an output based on a new input. Unsupervised ML evaluates data in terms of traits and uses the traits to form clusters of items that are similar to each other. Time series forecasting is the process of using a model to generate predictions (forecasts) for future events based on known past events [262, 75]. There are several machine learning methods: regression, classification, clustering [126], dimensionality reduction, ensemble methods, neural nets and deep learning [109], transfer learning, reinforcement learning, Natural Language Processing (NLP), word embeddings, etc. Regression is one of the predictive modeling techniques which analyzes the correlations between a target and independent variables. It is used to predict model time series and then find the causal effect correlations among different factors [263].
6.2.3.2.1 Performance Metrics  We considered the following error and accuracy metrics to evaluate our forecasting model. The equation of the performance metrics are summarized in Table 6.9.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Equation</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean Absolute Percentage Error (MAPE)</td>
<td>$MAPE = \frac{100%}{N} \sum_{i=1}^{N} \frac{</td>
<td>P_i - X_i</td>
</tr>
<tr>
<td>Direction Accuracy (DAC)</td>
<td>$DAC = \frac{1}{N}\left[\text{count}(\text{sign}(X_{cu} - X_{pr})) = \text{sign}(P_{cu} - P_{pr})]\right]$</td>
<td>$X_{cu}$ = Actual Current, $X_{pr}$ = Actual Previous, $P_{cu}$ = Predicted Current, $P_{pr}$ = Predicted Previous, $T_{pr}$ = Target Previous</td>
</tr>
<tr>
<td>Relative Absolute Error (RAE)</td>
<td>$RAE = \frac{\sum_{i=1}^{N}</td>
<td>P_i - X_i</td>
</tr>
<tr>
<td>Root Relative Squared Error (RRSE)</td>
<td>$RRSE = \sqrt{\frac{\sum_{i=1}^{N}(P_i - X_i)^2}{\sum_{i=1}^{N}(T_{pr} - X_i)^2}}$</td>
<td></td>
</tr>
<tr>
<td>Root Mean Squared Error (RMSE)</td>
<td>$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(P_i - X_i)^2}{N}}$</td>
<td></td>
</tr>
</tbody>
</table>

- **Mean Absolute Percentage Error (MAPE):** The Mean Squared Error (MSE) is the average of the square of the forecast error. As the square of the errors are taken, the effect is that larger errors have more weight on the score. In this case, MAPE comes in handy where the data are not understood from the error measure itself. As MAPE is a percentage error, it gives a good idea of the relative error. MAPE is the most widely used unit-free measure.

- **Direction accuracy:** DAC is a measure of prediction accuracy of a forecasting method in statistics. It compares the forecast direction (upward or downward) to the actual realized direction. DAC is similar to a binary evaluation. The metric only considers the upward or downward direction in the time series and is independent of the quantitative value of the increase or decrease. DAC provides the probability that the forecasting method can detect the correct direction of the time series [264].

- **Relative Absolute Error (RAE):** Relative measures give an indication of how well the forecaster’s predictions are doing compared to just using the last known target value as the prediction. For RAE, the comparison is done with respect to absolute error with the last known target. RAE takes the total absolute error and normalizes it by dividing by the total absolute error of the simple predictor (i.e., the last known target).

- **Root Relative Squared Error (RRSE):** RRSE is similar to RAE. The difference is that RRSE
takes the square root of the total squared error and normalizes it by dividing by the square root of the total squared error of the simple predictor (i.e., the last known target).

- **Root Mean Squared Error (RMSE):** MSE is more vulnerable to outliers since it gives extra weight to large errors. In addition, the squared errors are on different scale from the original data. Thus, RMSE, which is the square root of MSE, is often preferred to MSE as it is on the same scale as the data. However, RMSE is also sensitive to forecasting outliers. Researchers now seem to prefer unit-free measures for comparing methods [265, 266].

### 6.2.3.2.2 Machine Learning Algorithms

We use the following well-known machine learning regression algorithms to build our forecasting model classifier.

- **Support Vector Machine (SVM) for regression:** The ability of SVM to solve nonlinear regression estimation problems makes SVM successful in time series forecasting. SVM regression acknowledges the presence of non-linearity in the data and provides a proficient prediction model. In the SVM regression method, the sequential minimal optimization algorithm from Alex Smola and Bernhard Scholkopf [267] is used. The utilization helps to replace all the missing values and transform the nominal attributes to binary values. It also helps to normalize the attributes by default values.

- **k-Nearest Neighbor (KNN):** KNN works by storing the entire training dataset and querying it to locate the k most similar training patterns when making a prediction. KNN regression calculates the average of the numerical target of the K nearest neighbors [268]. KNN regression uses the same distance functions as KNN classification. KNN is better than LR when the data have less noise. KNN requires much less hyperparameter tuning compared to MLP.

- **Linear Regression (LR):** Linear regression is a statistic approach to model the correlations between the scalar dependent variable Y and single or several explanatory variables denoted x. In this regression approach, linear predictor functions are used to model the correlations and the unknown parameters of the functions are evaluated by the data [269]. Decision trees support non-linearity, whereas LR only supports linear solutions.

- **Random Forest (RF):** RF regression contains several decision trees and targets the class that is the mode of the classes' target by individual trees. The number of trees to be grown in the forest and the quantity of features or variables chosen at every node to develop a tree are the two standard parameters [270].
• **M5 model tree:** M5 tree is a decision tree learner for regression problems. The M5 algorithm assigns linear regression functions at the terminal nodes and fits a multivariate linear regression model to each subspace by classifying or dividing the whole data space into several subspaces. The M5 tree method deals with continuous class problems instead of discrete classes and can handle tasks with very high dimensions. It reveals piece-wise information of each linear model constructed to approximate nonlinear relationships of the dataset [271].

• **Gaussian process for regression:** The Gaussian algorithm is a very useful tool in non-linear multiple variate interpolation. It belongs to a kind of statistical framework in which observations happen in the continuous domain. During the process, each point of a certain continuous inputting space is related to the generally distributed random variable [272].

• **Multilayer Perceptron (MLP):** MLP is also known as Artificial Neural Networks (ANN). Neural networks are a complex algorithm to use for predictive modeling. This algorithm is inspired by a model of biological neural networks in the brain where small processing units called neurons are organized into layers that if configured well are capable of approximating any function. In regression problems, the interest lies in approximating a function that best fits the real value output [263]. MLP needs large training data compared to LR model and becomes slow due to its complex structure.

### 6.2.4 Methodology

In this section, we describe our Twitter dataset and the collection procedure. We mention our data pre-processing steps for sentiment analysis on two different datasets. We also discuss our machine learning forecast model for COVID-19 vaccination in the USA by presenting the methods for feature generation, training, and testing. We also provide the computational tools and environment in this section. We have made the collected corpus of tweets and the full source code publicly available ([https://github.com/nawsafirin/covid-19](https://github.com/nawsafirin/covid-19)) [June 15 2021].

#### 6.2.4.1 Computational Tools/Libraries

We used different Python libraries to do the sentiment analysis. For the collection of Twitter data and scraping tweets, we used Tweepy [273]. For natural language processing (pre-processing tweets before doing sentiment analysis), we used NLTK [274]. We used both TextBlob and VADER to do the sentiment analysis.

For building our machine learning forecast model, we used WEKA 3.8.3 [275] for the prediction of vaccinated US population.
6.2.4.2 Environment

Experiments in this study were performed on a personal computer with the configurations of Intel Core i7-4770 CPU 3.40 GHz × 8 Processor, 16 GB of RAM, 1 TB hard disk, 64-bit Windows 10 OS, and Ubuntu 16.04 LTS.

6.2.4.3 Sentiment Analysis

In this section, we describe the methodology of our work to perform sentiment analysis on Twitter data related to COVID-19 vaccination. Figure 6.6 portrays the schematic diagram for different steps of our sentiment analysis method on COVID-19 vaccination related tweets.

6.2.4.3.1 Twitter Data Collection  We used the Twitter API [276] to collect around 1.2 million original tweets using the Python library Tweepy [273]. We filtered the tweets by keywords associated with different COVID-19 vaccines and keywords mentioning a safe, healthy lifestyle after vaccination. Details about the keywords are mentioned in Tables 6.10 and 6.11. The tweets were collected over a five-week period beginning on April 10, 2021 and ending on May 17, 2021. We discarded the retweets during this time period with the Twitter API filter. We also only collected tweets in English we used NLTK for further data analysis. Our Twitter data collection pipeline is shown in Figure 6.7. Twitter’s API provides access to 1% of the public tweets by random sampling in near real time. Although questions might arise regarding biased or imbalanced data for collecting just 1% sample from all tweets, it has been shown that sentiments found from the samples of tweets obtained via the API and the full tweet dataset reflect the same sentiment percentage with very little deviation (<1.8%) [277]. In compliance with the Twitter content redistribution policy [278], we only made the tweet IDs publicly available corresponding to the collected tweet text used in this work.

We categorized the collected tweets into two different datasets.

- **Twitter Dataset 1:** The first dataset has all the tweets related to different vaccines. In Table 6.10, we show the total number of tweets collected for each of the vaccines. We also list the keywords that we used to scrap those tweets. We also mention the unique number of tweets based on different pre-processing steps of the Twitter data done before the sentiment analysis.

- **Twitter Dataset 2:** We selected five topics related to maintaining safety/precautions in response to COVID-19 pandemic. We selected the precautionary actions for individuals that help reduce virus transmission during pandemics including social distancing, wearing face masks, hand hygiene, and restricting interpersonal contact to outdoor settings [247]. Additionally, we added another topic related
to “travel” that was very restricted during the pandemic. For Twitter Dataset 2, we aimed to collect those tweets that have been shared by vaccinated people about their lifestyle after getting vaccines. Thus, we used the keywords \{vaccine, vaccination, vaccinated, shot, fully vaccinated, first dose, second dose, 1st dose, 2nd dose\} to scrap those tweets along with the keywords for particular topics, as listed in Table 6.11.

### 6.2.4.3.2 Pre-Processing of Data

- **Data Cleaning**: We removed the urls, punctuation marks, and special characters in this step.
- **Tokenization**: Text is divided into words (the smallest unit) in this step.
- **Stopwords Removal**: Some words such as “and”, “but”, “so”, and others are frequently used in the text but are not useful in the analysis. We do not use the predefined stopwords from any libraries.
Table 6.10: Twitter Dataset 1: Vaccination tweets used in our experimental evaluation.

<table>
<thead>
<tr>
<th>Vaccine</th>
<th>Keywords</th>
<th>Total Collected Tweets</th>
<th>Unique Tweets for Analysis</th>
<th>Unique Raw Tweets (Cleaned)</th>
<th>Unique Tweets (Stopwords Removed)</th>
<th>Unique Tweets (Normalized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pfizer</td>
<td>pfizer, Pfizer-BioNTech, BioNTech</td>
<td>681,660</td>
<td>580,126</td>
<td>521,186</td>
<td>496,050</td>
<td>493,755</td>
</tr>
<tr>
<td>Moderna</td>
<td>Moderna, moderna_tx, Moderna-NIAID, NIAID</td>
<td>301,782</td>
<td>292,965</td>
<td>271,182</td>
<td>258,141</td>
<td>257,092</td>
</tr>
<tr>
<td>Johnson &amp; Johnson</td>
<td>Johnson, Johnson and Janssen, Janssen Pharma-</td>
<td>17,709</td>
<td>16,104</td>
<td>12,994</td>
<td>12,711</td>
<td>12,703</td>
</tr>
<tr>
<td></td>
<td>ceutical, J&amp;J OXFORD VACCINE, Oxford-Astraeneca, Astraeneca, AstraZene, Covishield</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oxford-AstraZeneca</td>
<td>Sputnik V, sputnikv, sputnikvac</td>
<td>10,740</td>
<td>10,223</td>
<td>8,388</td>
<td>8,147</td>
<td>8,140</td>
</tr>
<tr>
<td>SputnikV</td>
<td>covaxin, Bharat-Biotech</td>
<td>41,136</td>
<td>37,804</td>
<td>33,175</td>
<td>31,809</td>
<td>31,697</td>
</tr>
<tr>
<td>Covaxin</td>
<td>coronavac, sinovac</td>
<td>21,947</td>
<td>20,591</td>
<td>18,621</td>
<td>18,095</td>
<td>18,060</td>
</tr>
</tbody>
</table>

because “not” or similar negative words, if removed, would change the sentiment of the sentence completely. Thus, we used our own list of stopwords that we made by modifying the most comprehensive collection of stopwords for the English language [279, 280]. We removed all negative words from this mentioned list so that it does not impact sentiment analysis.

- **Data Normalization:**
  - **Stemming:** In this step, we normalized the words by truncating the words to their stem words. We used Porter Stemmer from the NLTK library.
  - **Lemmatization:** Next, we lemmatized words to get the root words according to the part of speech.
<table>
<thead>
<tr>
<th>Tweet Topic</th>
<th>Keywords</th>
<th>Total Collected Tweets</th>
<th>Unique Tweets for Analysis</th>
<th>Unique Raw Tweets (Cleaned)</th>
<th>Unique Tweets (Stopwords Removed)</th>
<th>Unique Tweets (Normalized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hygiene</td>
<td>hand sanitizer, sanitizer, wash hands, wash face, soap, soap water, hand soap, sanitize mask, wearamask, masking, N95, face cover, face covering, face covered, mouth cover, mouth covering, mouth covered, nose cover, nose covering, nose covered, cover your face, covetyourface travel, outing, camping, air-travel social distancing,</td>
<td>99</td>
<td>84</td>
<td>84</td>
<td>83</td>
<td>83</td>
</tr>
<tr>
<td>Wear Mask</td>
<td></td>
<td>2,962</td>
<td>2,954</td>
<td>2,862</td>
<td>2,852</td>
<td>2,848</td>
</tr>
<tr>
<td>Travel</td>
<td></td>
<td>1,132</td>
<td>1,126</td>
<td>1,089</td>
<td>1,086</td>
<td>1,086</td>
</tr>
<tr>
<td>Social Distancing</td>
<td></td>
<td>727</td>
<td>726</td>
<td>707</td>
<td>705</td>
<td>705</td>
</tr>
<tr>
<td>Social Gathering</td>
<td></td>
<td>622</td>
<td>622</td>
<td>601</td>
<td>601</td>
<td>600</td>
</tr>
</tbody>
</table>

6.2.4.3.3 Sentiment Categorization We used two sentiment analysis tools, TextBlob and VADER, to get the sentiment of the tweets.

Sentiment analysis provides the polarity of the text. It also classifies text into positive and negative classes. Classification task can be performed by using different algorithms. Sentiment analysis methods can be machine learning-based or lexicon-based. In machine learning methods, a labeled dataset is required where the polarity/class of the text is already known. Lexicon is a collection of the predefined words where a polarity score is associated with each word. It is the easiest approach for sentiment classification. This classifier makes use of a lexicon and performs word matching which is used to categorize a sentence. In our work, both TextBlob and VADER use this lexicon-based approach.
In TextBlob, the sentiment category of each tweet is based on the polarity score of the text. For VADER, the classification is done based on the compound score.

We also generated the word clouds to visualize the important words based on the frequency of the words initially. However, this failed to provide useful information. Thus, we later used log-likelihood ratio \[281, 282, 283, 284\] to generate the word clouds. We used Equation (6.3) to calculate the log-likelihood ratio for each sentiment category (positive, negative, and neutral) . Here, \(w_i\) denotes word, \(x\) denotes one of the sentiment categories, \(\bar{x}\) denotes rest of the sentiment categories.

\[
q = \log \frac{P(w_i|x)}{P(w_i|\bar{x})} \tag{6.3}
\]

6.2.4.4 Forecasting Model for COVID-19 Vaccination in the USA

We used WEKA for building our forecasting model classifier. We chose WEKA as it is more user-friendly. Another reason is that the saved model can be used by general users (health or government officials) who want to use this model for prediction. We used the WEKA’s time series framework to build our model [262]. The workflow for our forecasting method is shown in Figure 6.8. We used the well-known machine learning regression algorithms: SVM, KNN, LR, RF, M5 Tree, Gaussian regression, and MLP. Among all of these algorithms, we chose the algorithm that performs best with less error. According to the authors of [285, 286, 287], SVM often works best for time series forecasting, and we also obtained similar findings, as described in Section 6.2.5.3.

![Figure 6.8: Workflow for COVID-19 vaccination forecasting model using WEKA.](image-url)

6.2.4.4.1 Vaccination Dataset We used the publicly available COVID-19 vaccination dataset by Our World in Data [288], widely used by journalists, policymakers, WHO, researchers, and the public. We used the data from the months of March, April and May (26 May) as known past events to predict the future.
We did not take data before March into consideration because the vaccine was very limited to certain people at that time.

Table 6.12: Features related to COVID-19 vaccination.

<table>
<thead>
<tr>
<th>No.</th>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>total_vaccinations</td>
<td>Total number of COVID-19 vaccination doses administered</td>
</tr>
<tr>
<td>b</td>
<td>people_vaccinated</td>
<td>Total number of people who received at least one vaccine dose</td>
</tr>
<tr>
<td>c</td>
<td>people_fully_vaccinated</td>
<td>Total number of people who received all doses prescribed by the vaccination protocol</td>
</tr>
<tr>
<td>d</td>
<td>new_vaccinations</td>
<td>New COVID-19 vaccination doses administered (only calculated for consecutive days)</td>
</tr>
<tr>
<td>e</td>
<td>new_vaccinations_smoothed</td>
<td>New COVID-19 vaccination doses administered (7-day smoothed) [For countries that do not report vaccination data on a daily basis, it is assumed that vaccination changed equally on a daily basis over any periods in which no data were reported. This produces a complete series of daily figures, which is then averaged over a rolling 7-day window.]</td>
</tr>
<tr>
<td>f</td>
<td>total_vaccinations_per_hundred</td>
<td>Total number of COVID-19 vaccination doses administered per 100 people in the total population</td>
</tr>
<tr>
<td>g</td>
<td>people_vaccinated_per_hundred</td>
<td>Total number of people who received at least one vaccine dose per 100 people in the total population</td>
</tr>
</tbody>
</table>
6.2.4.4.2 Feature Selection  The vaccination dataset contains 59 different attributes. Descriptions of all the attributes can be found in [289]. We do not discuss all attributes here because they are not all relevant to the forecasting of vaccinations, for example, total_cases, new_cases, new_deaths, icu_patients, population, etc. Out of 59 different attributes, we identified the attributes given in Table 6.12 as those related to vaccination and interesting to forecast. Of these nine attributes, we do not consider the smoothed attributes [e,i] for further analysis because the USA the vaccination data are reported on a daily basis. We only selected the two attributes people_vaccinated and people_fully_vaccinated for prediction of the partially and fully vaccinated US population and obtain an overview of the vaccination scenario in the USA.

6.2.4.4.3 Time Series Forecasting Classifier  For the time series forecasting model, we needed to set the following parameters.

- **Forecast Time Unit:** We used two different time units, for the prediction of Fully Vaccinated Population. The shorter time period is 1 week (7 days) and the longer time period is 2 months (60 days). By definition, parameter values of 7 days indicate that we are setting up our system to forecast 7 days ahead from the last date of the input training dataset. For the prediction of partially vaccinated population, along with these two units, we also predict a middle timeline, that is 39 days, to get the prediction for 4 July, the Independence Day of the USA.

- **Time stamp:** We have a “Date” attribute in our dataset. Date was set for the “Time Stamp” parameter of our system.

- **Periodicity:** Periodicity allows the user to specify the periodicity of the data. As we predicted the vaccinated population on a daily basis, we chose “Daily” for this parameter value.
• **Skip list:** For daily forecasting, it might happen that the entity remains closed on certain days or holidays, for example, businesses, the stock market, etc. For such entities, these time periods do not count. In such cases, “Skip list” is used to supply these types of time periods that are not to be considered. For our dataset, we do not have any such time period for which the regular operation does not continue. Thus, we do not need to set the “Skip List” parameter.

• **Confidence intervals:** This parameter is used for the system to compute confidence bounds on the predictions that it makes. We used 95% as the parameter value, which is also the default value of the parameter. The 95% confidence level means that 95% of the true target values fell within the interval. The confidence intervals were computed for each step-ahead level independently.

### 6.2.4.4.4 Selection of ML Algorithms
In this step, we changed the values of user-defined parameters associated with the different ML algorithms based on some statistical criteria and some trial and error procedures. The values of the parameters can be found in our code-base. Then, we evaluated each algorithm based on the performance metric and identify the best methods with less error.

### 6.2.5 Experimental Evaluation
In this section, we summarize the results from our experimentation. In Sections 6.2.5.1 and 6.2.5.2, we describe the overview of public sentiments related to COVID-19 vaccination. In Section 6.2.5.3, we describe the forecast result found from our model classifier.

#### 6.2.5.1 Public Sentiment on Vaccination

In this section, we show the sentiment analysis for Twitter Dataset 1.

##### 6.2.5.1.1 Vaccination Tweets by location
We are interested to know from which locations the tweets have been made. According to the geographical distribution of the tweets, we found the top countries. The tweets that do not mention a location mentioned are labeled as “Unknown” source. In Figure 6.10, we can see that the top countries include USA Pfizer, Moderna and Johnson & Johnson), India (for Sputnik V and Covaxin), and UK (for Oxford-AstraZeneca).

##### 6.2.5.1.2 Sentiment Distribution for Vaccination Tweets
We show the distribution of positive, neutral, and negative sentiment for different vaccines in Figure 6.9. Unless we pre-process data, many unimportant content remains in the analysis and misleads the result. Although in the raw data positive
sentiment is more prevalent than neutral sentiment for all of the vaccines, after pre-processing, we observe that many neutral text contents were classified as positive sentiments in the raw tweets. From the subjectivity scores of the different vaccines, given in Figure 6.12, we see that, for lower subjectivity scores, the tweets are more prone to factual information, which leads to more neutral tweets. The polarity score distribution of the tweets for different vaccines is shown in Figure 6.11. Although in the collected tweet dataset the majority of the tweets are neutral, the positive sentiment is higher than that of the negative one. It provides a relief that, instead of some adversarial effects of some of the vaccines, people are taking the vaccination positively in the battle with COVID-19. For all of the vaccines, positive sentiment is 20–25%, negative sentiment is around 10%, and the rest is neutral.

6.2.5.1.3 Sentiment Timeline for Vaccination Tweets  We show a day-to-day sentiment analysis of the tweets to have an understanding of the distribution of the tweets by date. This helps us to learn about some important topics on a particular day. Suppose, for the Johnson & Johnson vaccine (Figures
6.14c and 6.15c), we see a high threshold of tweets on 13 April 2021. This high peak is due to the tweets about blood clots from taking the Johnson & Johnson vaccine and related news to this topic. We show the distribution using the results both TextBlob and VADER to get a better comprehension of which sentiment analysis tool is working well for these Twitter datasets. Comparing Figures 6.14 and 6.15, we see that using both tools we have a similar trend. We observe from the following plots that VADER has the tendency to classify some neutral tweets into negative or positive ones and thus lessening the number of neutral tweets. In some cases (Figures 6.15c,d,f), the negative sentiment overrides the positive sentiments to a great extent. To verify such dissimilarities, we manually reviewed some tweets for the Johnson & Johnson vaccine dated 13 April 2021 where VADER (Figure 6.15c) shows a high peak of negative sentiment compared to TextBlob (Figure 6.14c) and listed some example tweets in Table 6.13. We observe that many tweets of positive and neutral sentiment as given in TextBlob are classified negative in VADER. Thus, for all of our other analyses regarding the sentiment analysis results, we mention the results found using TextBlob.
6.2.5.1.4 Top Frequency Words (Positive, Negative, Neutral) for Vaccination Tweets

In Figure 6.16, we show the high frequency word distribution for each vaccine. The words are categorized into negative, neutral and positive. Initially, we used the naive approach to generate the word clouds based on word frequencies, but the naive version does not reflect the significant results shown in Figure 6.13b (Moderna vaccine). In this case, positive, negative, and neutral, all three categories, portray the same words and do not capture any useful information (see the Appendix for details). Later, using the log-likelihood values, we obtained more relevant information, as given in Figure 6.16, for each of the vaccines. We obtained the most interesting findings in the negative category. For both the Johnson & Johnson vaccine and the Oxford-AstraZeneca vaccine, “Blood Clot” was pre-dominant in the negative category. Side effects of both of these vaccines [290] were propagated through tweets very rapidly, which is one of the advantages of social
media that can be used by health officials. We also found that many people have fever and headache after taking the Moderna vaccine, as reflected in Figure 6.16b as the side effects of the Moderna vaccine. Thus, we found the side effects of these different vaccines at a glance from the generated word clouds.

6.2.5.2 Public Sentiment on Healthy Lifestyle after Vaccination

In this section, we show the sentiment analysis for Twitter Dataset 2.

6.2.5.2.1 Post-Vaccination Tweets on Healthy Lifestyle by location

Figure 6.10h shows that most tweets were generated in USA, India, and Canada, representing 34%, 15%, and 11% of the total tweets, respectively.
Figure 6.13: Top Frequency Word Distribution for different Vaccines: Word Cloud generated using word frequencies
6.2.5.2.2 Sentiment Distribution for Post-Vaccination Tweets on Healthy Lifestyle

We show the distribution of positive, neutral, and negative sentiment for different post-vaccination tweets in Figure 6.17. Pre-processing of data is necessary, as explained in Section 6.2.5.1.2.

For Twitter Dataset 2, we had the same observation. Only for the topic “social distancing” we have very few tweets with neutral sentiment, as given in Figure 6.17d. We provide a reasoning for such observation from the subjectivity score of the tweets given in Figure 6.23. In the case of tweets on “social distancing”, the distribution of subjectivity score is almost equally between fact and opinion. However, in the other four topics, the factual tweets overpower the judgemental tweets that leads to more neutral sentiment. We have shown the polarity score distribution of the tweets for Twitter dataset 2 in Figure 6.22. Observing Figure 6.17a–e, we see that, after vaccination, people are still conscious about maintaining a healthy lifestyle given around 70% and 40% of positive sentiment in maintaining social distancing and hygiene, respectively. The
negative sentiment is 50% and 35% less for “social distancing” and “hygiene” topics, respectively. In the case of using mask, people have a moderate outlook.

We see around 30% positive sentiment for wearing a mask and the negative sentiment is half that of the positive one. Vaccinated people have become more open to travel and social gatherings, as reflected by around 5% difference between positive and negative sentiments. We are optimistic of the fact that negative sentiments do not override positive sentiments for any of the topics related to maintaining a healthy lifestyle after vaccination.

6.2.5.2.3 Sentiment Timeline for Vaccination Tweets Observing Figures 6.18 and 6.19, we can see some dissimilarities. As described in Section 6.2.5.1.3, VADER shows some misclassification for Twitter Dataset 2 as well. We manually inspected some tweets, and some misclassified example tweets with the sentiment are given in Table 6.14.
6.2.5.2.4 Top Frequency Words (Positive, Negative, Neutral) for Post-Vaccination Tweets on Healthy Lifestyle

In Figure 6.20, we show the high frequency word distribution for each topic, categorized into negative, neutral, and positive. Using word frequencies, no useful information can be observed, as shown in Figure 6.24. Later, using the log-likelihood values, we could better capture information for the positive words such as “stay, safe, well, happy, party, thank,” as given in Figure 6.20. In Figure 6.20c, we see the idea...
of travel ban in India and travel risk in Florida. However, the negative words are still scattered for other topics. Generating word clouds combining all topics provides a better overview of the negative words such as “ban, sick, stop, emergency, death, mask, risk” given in Figure 6.21. In Figure 6.21, at a glance, we can see that people are more reluctant to “mask” and lean towards “party” after getting vaccinated.
Figure 6.17: Public Sentiment Distribution for Twitter Dataset 2 on Healthy Lifestyle

6.2.5.3 COVID-19 Vaccination Forecast for the USA

We used different machine learning models to forecast the fully vaccinated population of the USA. We also predicted the percent of the population that will be vaccinated with at least one dose of COVID-19 vaccine. The results are described in the subsequent sections.

178
6.2.5.3.1 Fully Vaccinated Population Prediction  The predicted outputs for a period of 60 days (2 months) from our model classifier are shown in Figure 6.25.

We also performed a prediction for a shorter period of 7 days. For brevity, the predicted output is shown in Figure 6.27. The prediction accuracy and error of different models are given in Table 6.15. (We omit the values for RMSE while discussing our result because RMSE represents unit error instead of percentage error. We did not further process and convert RMSE to percentage error because the other metrics show relative representation as RMSE.) For the near-time forecast, SVM, M5 model tree, and MLP all have 100% DAC. Although RF has the lowest MAPE, RAE, and RRSE, the DAC is below 100%. The population of fully vaccinated people being an aggregation metric will always increase with time and so we prioritize the models having the highest DAC over minimum error metrics. Thus, we chose MLP model with 100% DAC and the lowest error among SVM, M5 model tree and MLP models.

For the 2-month prediction period, all models except KNN and LR have 100% DAC. RF again has least errors, but Figure 6.25d shows that the predicted output is almost flat over the predicted time period. Considering the aggregation metric, we expect the output to increase over time, and thus selected a different model. Of all other methods, SVM has the lower error and is considered as the best classifier for the 2-month forecasting. Thus, using SVM, we can predict that by 25 July 2021 around 48% of the US population will be fully vaccinated (2 doses as applicable according to the vaccine) against COVID-19.
Table 6.14: Random tweets from Twitter Dataset 2 on social distancing.

<table>
<thead>
<tr>
<th>Original Tweet</th>
<th>TextBlob</th>
<th>VADER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Today marks my two weeks post second dose, so I am fully vaccinated. Still going to do the thing and wear my mask and social distance, though. I never know what difficulties people around me may have, so I would be mortified if I caught this again and spread it to someone else.</td>
<td>positive</td>
<td>negative</td>
</tr>
<tr>
<td>Finally got the first Dose of #covid19 Vaccine. Got Vaccinated We will still wear mask and practice social distancing. Lets fight together against Covid. #staysafeandkeepsafe #gotvaccine #vaccinesafety #Vaccination2021</td>
<td>positive</td>
<td>negative</td>
</tr>
<tr>
<td>No. Frankly I’m not changing social distancing practice until two or three weeks after we’ve both had our second dose of vaccine. [June 5 2021]</td>
<td>positive</td>
<td>negative</td>
</tr>
<tr>
<td>Received my First Dose of COVID Vaccination. Urge all eligible(18-45 years)to get your shot soon at your nearest Vaccination Centres. Vaccination &amp; Social Distancing are the only few weapons to defeat the virus. #IndiaFightsCorona #We4Vaccine #CoWIN #VaccinationForAll #WearMasks [June 5 2021]</td>
<td>positive</td>
<td>negative</td>
</tr>
<tr>
<td>@MeidasTouch Received my second dose at the begging of the year. Had no secondary effects with first or second dose. I’m 67 and I’m still following guidelines, wearing a mask, social distancing, not going out into large group events.</td>
<td>positive</td>
<td>negative</td>
</tr>
</tbody>
</table>

6.2.5.3.2 Partially Vaccinated Population Prediction  The predicted outputs for partially vaccinated population of the USA for a period of 60 days (2 months) from our model classifier is shown in Figure 6.26. We also performed a prediction for the timeline 4 July 2021 to see the estimated percentage if it aligns with the US government expectations. The prediction accuracy and error for different models are given in Tables 6.16 and 6.17. Table 6.16 summarizes the result for the timeline of July 4, 2021. Table 6.17 summarizes the result for the 2 month time-frame. As for KNN and RF, the classifier does not work well, thus we do not mention the numbers in the table to keep it simple. For the 4 July forecast, SVM, M5 model tree, and MLP all modelshave 100% DAC. SVM has the lowest MAPE and RAE among these three models. Using SVM, the model predicts that around 57.62% of the US population will get at least one dose of vaccine. If we consider only the adult population [291], this percentage becomes 73.53%, which is nearly the expected percentage (70%) given by US President Joe Biden [292].
For the 2-month prediction period, all models have 100% DAC. Of all other methods, MLP has the lowest error and is considered as the best classifier for the 2-month forecasting. Thus, using MLP, we can predict that by 25 July 2021 around 80% of the US adult population will get at least one dose of vaccine against COVID-19.
Table 6.15: Prediction accuracy and error for different machine learning models to forecast fully vaccinated population in the USA.

<table>
<thead>
<tr>
<th>Forecast Time Unit</th>
<th>7 Day</th>
<th>2 Months</th>
<th>Predicted Output (millions)</th>
<th>Percentage of US Population (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML Models</td>
<td>RR-SE</td>
<td>RAE</td>
<td>MAPE</td>
<td>DAC</td>
</tr>
<tr>
<td>SVM</td>
<td>9.68</td>
<td>7.93</td>
<td>1.01</td>
<td>100</td>
</tr>
<tr>
<td>KNN</td>
<td>33.23</td>
<td>16.36</td>
<td>1.58</td>
<td>97.26</td>
</tr>
<tr>
<td>LR</td>
<td>22.26</td>
<td>19.36</td>
<td>2.19</td>
<td>98.63</td>
</tr>
<tr>
<td>RF</td>
<td>5.10</td>
<td>3.86</td>
<td>0.39</td>
<td>97.26</td>
</tr>
<tr>
<td>M5</td>
<td>12.55</td>
<td>10.69</td>
<td>1.18</td>
<td>100</td>
</tr>
<tr>
<td>Gaussian</td>
<td>97.87</td>
<td>89.26</td>
<td>10.59</td>
<td>83.78</td>
</tr>
<tr>
<td>MLP</td>
<td>8.34</td>
<td>7.17</td>
<td>0.74</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 6.16: Prediction accuracy and error for different machine learning models to forecast partially vaccinated population (at least one dose vaccine) in the USA for July 4, 2021.

<table>
<thead>
<tr>
<th>ML Models</th>
<th>RRSE</th>
<th>RAE</th>
<th>MAPE</th>
<th>DAC</th>
<th>Predicted Population (Millions)</th>
<th>Population (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM</td>
<td>1.15</td>
<td>1.03</td>
<td>0.38</td>
<td>100</td>
<td>190.7</td>
<td>57.62</td>
</tr>
<tr>
<td>LR</td>
<td>2.51</td>
<td>1.94</td>
<td>0.74</td>
<td>98.63</td>
<td>183.4</td>
<td>55.41</td>
</tr>
<tr>
<td>M5</td>
<td>3.66</td>
<td>3.41</td>
<td>1.24</td>
<td>100</td>
<td>206.8</td>
<td>62.48</td>
</tr>
<tr>
<td>Gaussian</td>
<td>15.07</td>
<td>13.93</td>
<td>5.17</td>
<td>82.93</td>
<td>271.3</td>
<td>81.97</td>
</tr>
<tr>
<td>MLP</td>
<td>1.24</td>
<td>1.04</td>
<td>0.39</td>
<td>100</td>
<td>193</td>
<td>58.32</td>
</tr>
</tbody>
</table>

Table 6.17: Prediction accuracy and error for different machine learning models to forecast partially vaccinated population (at least one dose vaccine) in the USA for 2 month time-frame.

<table>
<thead>
<tr>
<th>ML Models</th>
<th>RRSE</th>
<th>RAE</th>
<th>MAPE</th>
<th>DAC</th>
<th>Predicted Population (Millions)</th>
<th>Population (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM</td>
<td>0.77</td>
<td>0.63</td>
<td>0.33</td>
<td>100</td>
<td>202.7</td>
<td>61.24</td>
</tr>
<tr>
<td>LR</td>
<td>0.76</td>
<td>0.69</td>
<td>0.36</td>
<td>100</td>
<td>190.3</td>
<td>57.5</td>
</tr>
<tr>
<td>M5</td>
<td>2.29</td>
<td>1.96</td>
<td>1</td>
<td>100</td>
<td>227.4</td>
<td>68.71</td>
</tr>
<tr>
<td>Gaussian</td>
<td>7</td>
<td>6.02</td>
<td>3.08</td>
<td>85</td>
<td>340.4</td>
<td>102.85</td>
</tr>
<tr>
<td>MLP</td>
<td>0.6</td>
<td>0.54</td>
<td>0.28</td>
<td>100</td>
<td>206.7</td>
<td>62.44</td>
</tr>
</tbody>
</table>
6.2.6 Conclusions

This is a study of COVID-19 vaccination in the aftermath of the 2020 COVID-19 outbreak with quantitative analysis. We analyze public sentiment on different COVID-19 vaccines from Twitter dataset. We find that the public sentiment is more positive than negative despite some side effects found in some of the vaccines. Although the majority of the tweets (60–70%) are deemed neutral, the remaining part consists of mostly positive tweets (20–25%). We also do an analysis of the public’s healthy lifestyle after vaccination. We find that people are conscious about maintaining social distancing with positive sentiments of 70% where neutral sentiment was much less (5%) compared to higher neutral sentiments in the remaining cases. People are also positive about maintaining hygiene with 40% and 10% positive and negative sentiments respectively. Although we observe that people are more reluctant to using masks and more prone to travel or do social gathering after vaccination, there is less difference between the positive and negative sentiments. However,
Figure 6.23: Subjectivity Score of Public Sentiment for Twitter Dataset 2 on Health

it is a relief that the negative sentiments do not exceed the positive sentiments in any of the cases. We also show a time series forecast of the US population to be vaccinated for a time period of 2 months. Our vaccination forecast model predicts that around 62.44% of the total population will get at least one dose vaccine and 48% will be fully vaccinated by the end of July 2021. Our prediction model gives a similar estimate of having partial vaccination of the adults to be 73.53% that the US Government is projecting to be 70% on Independence Day (4 July 2021). The results of these analyses can be utilized to better comprehend Twitter users’ opinions about COVID-19 vaccination and their lifestyle after vaccination. The current discoveries give a steppingstone to measure the public’s conversation about COVID-19 vaccination and guidelines for a healthy way of life during the pandemic. This will help the public health authorities and policymakers to understand how individuals are endeavoring to adapt with their mental conditions during these extraordinary times and what services and resources should be made accessible to the public. This study will help health and government officials to better comprehend and plan vaccination campaigns. The constraints of this exploration are in the dataset size and time period during which it was collected. It would be interesting to have information from a longer time span to perceive how sentiments change over a longer time, especially when the pandemic comes to an end. Classifying tweets into different emotions, such as inspired, happy, annoyed, sad, angry, afraid, etc., to fully understand and reveal the sentiment of the tweets is also an interesting future direction.
Figure 6.24: Top Frequency Word Distribution for Twitter Dataset 2 on Health: Word Cloud generated using word frequencies
Figure 6.25: COVID-19 vaccination prediction to forecast fully vaccinated population in the USA for different machine learning models with forecast time of 2 months.
Figure 6.26: COVID-19 vaccination prediction to forecast partially vaccinated population in the USA for different machine learning models with forecast time of 2 months.
Figure 6.27: COVID-19 Vaccination Prediction for different Machine Learning Models with Forecast Time of 1 week
Chapter 7

Conclusion

In this chapter, we conclude the dissertation by summarizing our contributions. Then, we highlight how the work can be applied to several future research directions.

7.1 Summary of Contributions

Overall, in this dissertation we have applied different graph mining techniques on large scale networks with millions and billions of vertices and edges from different scientific domains (social, blog, communication, collaboration, citation, biological etc). The parallel algorithms we have designed scales well with large dataset and shows good performance. We also show scalability of deep neural networks’ model performance on graphs. We also include two real-world big graph applications using machine learning in this work. We present different parallel implementations of community detection algorithms for static and temporal networks. We have identified bottlenecks for different methods of designing parallel implementations and provide optimized solution. Our parallel algorithms achieve $12 \times$ speedup for static CD, $30 \times$ speedup for temporal CD, $4 \times$ performance gain on GPU using PyTorch with CUDA for GCN) via semi-supervised node classification. We have identified a subset of graph metrics to understand the evolving community structure quality for different temporal networks. We have designed a scalable solution to the Sparse DNN challenge with $4.7 \times$ speedup on GPU using data parallelism of TensorFlow. We have designed a machine learning classifier to detect webspam from webgraph and devised feature selection method from the graph data, an important application of graph mining on real-world network. We also perform sentiment analysis on social network, Twitter, and predict vaccinated population using timeseries forecasting. We have concluded with important insights on public attitude towards COVID-19 vaccinations.
7.2 Future Research Directions

In this dissertation, we have designed and implemented several parallel algorithms that can be applied in different scientific domains to solve certain specific problems. Our parallel community detection algorithms DPLAL/DyGMPCD can be applied to brain networks for finding hierarchical community structures within the brain’s functional network. How neural units cluster into densely interconnected groups can provide the coordinated activity-characteristic of perception, action, and adaptive behaviors. Any alteration indicates disease otherwise, healthy cognition. Recommendations for Reddit users can be provided to find similar subreddits based on general user behavior and in other recommendation systems using our parallel CD algorithms. Epidemic spreading is another important application of clustering where our parallel algorithms can be applied for efficiency to mitigate infectious diseases like Covid-19. Experiment with other load balancing schemes can be done to find more efficient load balancing scheme to make our parallel algorithms more scalable and optimized in future. The design of framework for model parallelism and hybrid parallel implementations for performance optimization of our deep neural models is another direction. In brief, we believe that our developed parallel algorithms and models will be useful to the broader research community for big data analysis and mining represented by graphs.
Bibliography


[143] “Datasets.”


[186] “Motivation.”


[228] R. Jain, “Introduction to naive bayes classification algorithm in python and r. [online] available at:”


[276] “What’s new with twitter api v2 | docs | twitter developer platform.”


[278] “Developer agreement and policy – twitter developers | twitter developer platform.”


[281] “Comparing word form counts.”


Vita

Naw Safrin Sattar

Naw Safrin Sattar is a PhD Candidate in the department of Computer Science, working as a Graduate Research Assistant in the Big Data and Scalable Computing Research Group under the supervision of Dr. Shaikh Arifuzzaman in the University of New Orleans. She joined the University of New Orleans in Fall 2017. She completed her Master’s in Computer Science from University of New Orleans in Spring 2019. She obtained her Bachelor’s degree in Computer Science and Engineering from Bangladesh University of Engineering and Technology (BUET) in 2016. Her research interests include Big Data Analytics, High Performance Computing, Graph Mining, Machine / Deep Learning, Parallel Algorithms, and Performance Optimization. Her PhD Dissertation is one of the top 10 dissertations selected by the leading conference SuperComputing (SC’21) to be showcased at the conference. She is one of the 15 recipients of the Parallel Computing Summer Research Internship at Los Alamos National Laboratory in Summer 2021. She received the Computing Sciences Research Pathways Fellowship in the student-faculty program from Lawrence Berkeley National Laboratory in Summer 2019. She has received several awards and grants starting from her undergraduate studies to attend several research conferences and workshops funded by NSF and other generous sponsors. She will be joining as a Postdoctoral Research Associate - AI Technologies for HPC Operational Data Analytics at the Oak Ridge National Laboratory, one of the world leading research labs after the completion of her PhD degree in Fall 2022. She aspires to be a full-fledged research professional in an organization, preferably a national lab or as a faculty member in the academia.