論文 / 著書情報
Article / Book Information

題目(和文)	大規模動的グラフ解析の性能最適化手法
Title(English)	Performance Optimization of Large-Scale Dynamic Graph Analytics
著者(和文)	金刺宏樹
Author(English)	Hiroki Kanezashi
出典(和文)	学位:博士(理学), 学位授与機関:東京工業大学, 報告番号:甲第11222号, 授与年月日:2019年6月30日, 学位の種別:課程博士, 審査員:松岡 聡,増原 英彦,遠藤 敏夫,脇田 建,額田 彰
Citation(English)	Degree:Doctor (Science), Conferring organization: Tokyo Institute of Technology, Report number:甲第11222号, Conferred date:2019/6/30, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Tokyo Institute of Technology

Ph.D. Thesis

Performance Optimization of Large-Scale Dynamic Graph Analytics

Advisor Prof. Satoshi Matsuoka

May 2019

Mathematical and Computing Sciences in the Graduate School of
Information Science and Engineering

Hiroki Kanezashi

Abstract

Large-scale graph data analysis has been necessary for real-world analytics. Discrete simulations with graph structures are also fundamental techniques to investigate and validate more detailed analytic results. Moreover, the scale of data has become much larger, and HPC systems with ample memory space and multiple processors and computation nodes are indispensable to conduct such analytics and simulations. The demand of large-scale simulation is increasing in order to extract global statistical outcomes, and city- or country-level simulations must be performed simultaneously even they assume these are local what-if simulation.

However, many performance issues happen such as load imbalances and communication overheads, which often prevent the scalability of parallel and distributed computations. These problems usually occur while running an application handling dynamic graph data. In an agent-based traffic application, for example, even we partition road network equally the number of agents for each partition becomes imbalanced and the overall application will not scale.

To address the problem, we proposed performance optimization methods for dynamic data management based on the graph structures. We set several goals of the performance optimizations: to gain scalability of parallel processing by the reduction of costs such as load imbalance and synchronization overhead, and use graph analytics for dynamic graph analytics with incremental processing and adaptive parameter optimizations.

First, we proposed the performance optimization method to adjust synchronization intervals among all threads and processes in a parallel and distributed agent-based traffic simulation. It skips synchronization phases and allows all the distributed simulation processes to run individually for most simulation steps. We also proposed an adaptive migration method for an agent-based simulation framework to re-assign agent objects to threads and remote processes. From the previous execution logs, it partitions the road networks, and then it automatically re-assigns computation processes and threads to cross points and vehicle agents while running simulations. We evaluated these proposed methods in parallel and distributed computation nodes of TSUBAME 2.5 supercomputer with real road network and commutation trip data in the Dublin city. The synchronization interval adjustment method achieves

up to 35% speed-up in more than four nodes, and dynamic agent object assignment method across computation nodes achieves 25% speed-up on average.

Second, in order to minimize the computation time of graph analysis for time-evolving graphs, we proposed incremental graph analytics algorithms.

We proposed an incremental community detection algorithm named IncrementalDEMON for time-evolving graphs more efficiently by extending the existing community detection algorithms to support incremental processing with vertex and edge additions or deletions. The IncrementalDEMON consists of three core functions to construct overlapping communities, and we added an incremental feature each of them. In the evaluation, we used time-evolving social networks with millions of edges with timestamps, and our incremental community detection algorithm performs up to 101 times speedup compared to the original algorithm.

We also proposed an incremental graph pattern matching (IGPM) algorithm as an extension of an existing approximate subgraph isomorphism algorithm, and parameter adjustment framework named Partial Execution Manager (PEM) with reinforcement learning to determine the vertex sets for re-computations. Similar to the IncrementalDEMON, IGPM algorithm consists of three core functions which are incrementally extended to find the most likely matching vertices and edges in the best effort. The PEM consists of graph clustering and reinforcement learning components. The graph clustering component detects the vertex sets for re-computation, and the reinforcement learning component adjusts the granularity of the graph clustering to find more matched patterns.

Our incremental graph pattern matching algorithm performs up to 10.1 times speedup compared to the original subgraph isomorphism algorithm, and our adaptive optimization framework achieves up to an additional 1.95 times speedup. Moreover, the adaptive and incremental algorithm found up to 73% additional patterns compared to the original algorithm.

Finally, we made qualitative evaluations and discussions for our proposed methods about adaptive parameter adjustment from environments and incremental graph algorithms such as the applicability and limitations for other types of real-world graphs, other graph algorithms, and agent-based simulations. We validated how these methods can apply to other real-world applications and graph datasets with several representative cases, and how each optimization method affects the precision of outputs such as the travel time of each vehicle in the traffic simulation and subgraph size distributions in the community detections and pattern matching.

This thesis presents some contributions to discover the movement of real-world with a graph structure. Our series of researches can apply to various fields of social data analytics applications with other graph algorithms and social simulations with more extensive and time-evolving graph data.

Acknowledgements

I would like to show appreciation to my supervisor Professor Satoshi Matsuoka for technical and research support. He always helped me to think deeply about my researches fields in various aspects. I learned many important things from him - not only technical knowledge but also the way of viewing problems and examining solutions critically from various angles, which are essential for research scientists.

I would like to express my gratitude to all the members of Matsuoka Laboratory. They assisted my researches by offering advice, helping me to conduct experiments in servers in laboratory and TSUBAME supercomputer.

I would like to appreciate Professor Toyotaro Suzumura, my previous supervisor in Tokyo Institute of Technology, for offering invaluable opportunity for discussing in Dublin city, giving useful information as well as traffic trip data and providing me a great number of advices about agent simulation. He also gave me invaluable opportunities to conduct my research about performance optimization of graph analytics in IBM Thomas J. Watson Research Center.

I also appreciate members of IBM research and University College of Dublin (UCD) for exchanging useful ideas for agent-based traffic simulation and graph processing optimizations.

I gratefully appreciate the financial support from Japan Science and Technology Agency (JST) for Graph CREST and Extreme Big-Data (EBD) crest, and Real-World Big-Data Computation Open Innovation Laboratory (RWBC-OIL). These supports are very helpful to continue my researches.

Finally, would like to express my appreciation to my families and relatives. They always encouraged and supported my researches. Without their support, I could never complete my thesis.

Hiroki Kanezashi, May 2019

Publications

Refereed Conference Paper

- Hiroki Kanezashi and Toyotaro Suzumura, "Performance optimization for agent-based traffic simulation by dynamic agent assignment." 2015 Winter Simulation Conference (WSC), Huntington Beach, CA, December 2015.
- Hiroki Kanezashi, Toyotaro Suzumura, Dario Garcia-Gasulla, Min-hwan Oh and Satoshi Matsuoka, "Adaptive Pattern Matching with Reinforcement Learning for Dynamic Graphs", 2018 IEEE 25th International Conference on High Performance Computing (HiPC), Bengaluru, India, December 2018.

Refereed Workshop Paper

- Hiroki Kanezashi and Toyotaro Suzumura. "An incremental local-first community detection method for dynamic graphs." Big Data (Big Data), 2016 IEEE International Conference on. IEEE, 2016.

Non-refereed Workshop Paper

- Hiroki Kanezashi, Toyotaro Suzumura and Satoshi Matsuoka, "Performance optimization of large-scale traffic simulation in parallel distributed systems." IPSJ SIG Technical Reports 2015-HPC-148 (28), Oita, Japan, March 2015.

Refereed Poster

- Hiroki Kanezashi, "Performance Optimization of Large-Scale Traffic Simulation on Parallel & Distributed Systems", International Supercomputing Conference (ISC'15) HPC in Asia Posters, Frankfurt, Germany, July 2015.

Contents

Chapter 1

Introduction

1.1 Motivation

Recently, demands and practical applications of complex network analysis techniques have been increasing to find peculiar patterns and characteristics of vertices and subgraphs from million- and billion-scale data.

Agent-based simulations are also fundamental techniques to extract knowledge from real-world large-scale graphs. For example, various traffic simulations have been proposed and conducted to find out the distribution of trip time or the affects of city planning (e.g. construct or close roads). City- or country-level simulations are required to extract more accurate outcomes because even small differences of simulation scenarios affect to wide range of road networks.

As of December 2014, the scale of social network at Facebook 1.39 billion users (vertices) and more than 400 billion follower-following relationships (edges) [1]. In 2010, Twitter already has 42 million users and 1.5 billion follower-following relationships. The million- and billion-scale networks appear in not only social network services, but also other web services such as question-answering and discussion forums, product co-purchasing networks in e-commerce such as Amazon [2], and online web-page editing such as Wikipedia Talk [3].

Besides the largeness of its size, the real-data also has time-evolving features. It frequently updates its structures and attributes second by second. For example, Twitter network growth from 2006 to 2012 is shown in Figure 1.1 [4]. The users of Twitter is rapidly increasing from 2009, and the growth speed is also increasing.

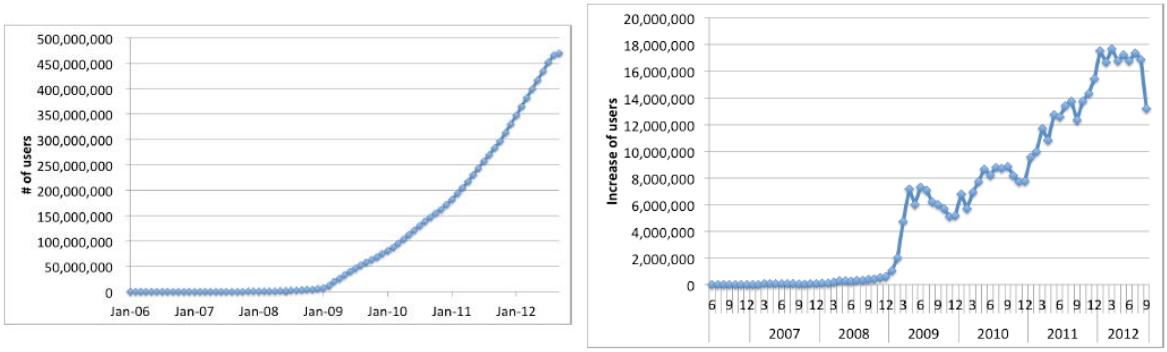


Figure 1.1: Transitions of the number of Twitter users (left) and monthly increase of users (right) from [4]. The number of users have been increasing, and the increase speeds up since the beginning of 2009.

Not only follower-following networks, but sending messages from one user to others directly and posting messages with a mention of other users such as retweets can form a dynamic network. Kwak et, al. crawled Twitter users, follower-following relationships, tweets and trending topics [5] and extracted top-20 users by the PageRank of in the follower-following networks and retweet diffusion networks.

Nowadays many graph analytics algorithms such as community detection and graph pattern matching have been proposed as increasing demands for extraction of knowledge from such large-scale and time-evolving networks. For example, community detection algorithms in social networks are often used to find out account groups with similar organizations, interests, and so on. Graph pattern matching algorithms are also used in social networks and personnel networks to extract certain personnel relationships [6].

Besides graph analytics algorithms, social simulations are also fundamental to find knowledge from large-scale data including graphs. Particularly, agent-based simulations can imitate fine-grained and detailed real-world behaviors. The input and output data of some agent-based simulations form graph structure. For example, a traffic simulator such as Megaffic [7] uses road network data and trip data to simulate the behaviors of each vehicle agent step-by-step.

For the graph data as simulation outputs, we construct money transfer network from log data of mobile money transfer simulator such as PaySim [8]. In this case, each bank account corresponds a vertex, and each transaction corresponds an edge with timestamps.

1.2 Problem Statement

While running graph analytics with large-scale and time-evolving attributed graph data, it encounters several problems related to the performance. Some problems are related to its scalability for high-performance computing environments with multiple computation nodes.

1.2.1 Load Imbalance in Parallel and Distributed Executions

It is hard to handle large geographical data with millions of cross points and roads as the road network and millions of vehicles as agents in a standalone computation system. In order to handle million- and billion-scale graphs, we have to partition the graph data into many small clusters and assign them to CPU threads, processes or computation nodes equally. However, data partitioning and clustering for complex networks is more difficult because graph clustering algorithms have different trade-offs.

1.2.2 Communication and Synchronization Cost

Related to the load imbalance problem, communication costs are significant in parallel and distributed executions. Even graph clustering process partitioned graphs into small subgraphs with the same number of vertices or edges, it causes large communication costs while executing parallel graph analytics or its simulations in distributed environment if these subgraphs have a number of edge-cuts. Large number of edge-cuts affects more frequent synchronizations and data copying between different threads and processes. Some options of METIS library [9] tries to minimize such edge-cuts, but it cannot divide complex graphs into small subgraphs with the similar size in general.

All processes and threads handling partitioned subgraphs must synchronize with neighboring processes and threads to keep consistency. However, when the edge-cuts are large and each subgraphs are neighboring to many other subgraphs, the synchronization frequency becomes larger and an extra communication overhead emerges. Moreover, when some threads and sub-processes take longer time to process vertices and edges in the assigned subgraphs than others, all neighboring subgraphs have to wait for the finish. The load imbalance affects to the synchronization problems and weakens the scalability by threads and processes.

1.2.3 Computational Cost for Dynamic Graphs

Most real-world networks have time-evolving features. The structure are frequently updated by adding and removing vertices and edges or updating these attributes. Some applications require real-time analytics result outputs according to the dynamics of the graphs.

For example, banks need to find suspicious subgraph patterns from account-to-account transaction networks. In this case, it is time-wasteful and not realistic to run graph pattern matching algorithm from scratch seconds to seconds because the transaction network size is more than one billion vertices and one trillion edges, and most of graph structure does not change. Considering this situation, we need more sophisticated algorithms to minimize computation costs with keeping precision.

In a traffic simulation, on the other hand, the road network structure itself does not frequently change, but the edge attributes (road status) such as the number of vehicles running and whether the road is closed or not due to an accident. In this case, the simulator has to update the assignments of threads and processes to cross points, roads and vehicles adaptively to balance these workloads and minimize the synchronization latency and communication costs.

However, the efficiency of incremental processing sometimes decreases due to the changes of many vertices and edges, and the level of changes are not always stable during the execution. When many vertices and edges are updated, it needs whole updates of current results. On the other hand, when a few vertices and edges are added or deleted, it only has to update subgraphs around updated area.

The adaptive re-assignment method itself also causes the extra computation and communication costs because it has to migrate many agent vertices to another processes at the time. For that reason, we need to consider the trade-off between the costs for temporal computation and communication and the costs of load imbalance and communication overhead for long time.

1.3 Approaches and Proposal

To overcome these problems related to the large-scale and time-evolving graph analytics and its applications, we propose performance the following optimization methods; **Adaptive Re-assignment of Computational Resources and Objects**, **Incremental Graph Algorithms** and **Adaptive Parameter Adjustments**.

1.3.1 Adaptive Reassignment of Computational Resources and Objects

We introduce an optimization method to reduce workload imbalance among processes and threads in a distributed agent-based simulation to gain the scalability of parallel executions. This method consists of two steps; First, it generates the mapping file to bind cross points to Java threads and processes (computation nodes) from the distribution of running vehicles in the previous simulation log snapshots. Then, it re-assigns vehicle agent object to threads and processes periodically.

1.3.2 Incremental Graph Algorithms

To minimize the re-computation costs, we propose incremental graph analytics for dynamic graph analysis.

First, we proposed an incremental community detection algorithm based on overlapping community detection algorithm named DEMON. DEMON consists of three core functions to build up communities with the bottom-up approach, and our incremental method also consists of incremental version of these core functions.

Next, we proposed an incremental graph pattern matching algorithm. It is also based on an existing graph pattern matching algorithm named G-Ray, which also consists of three core functions to extract the best patterns.

Because our proposed incremental methods are extensions of existing community detection and graph pattern matching algorithms, we also make discussion whether our incremental algorithms are applicable to other community detection, graph pattern matching, and different graph analytics.

1.3.3 Adaptive Parameter Adjustments

The structure of time-evolving graphs drastically updates with the

When the graph structure has drastically changed from the original state, the initial incremental graph algorithms or these parameters do not optimize these algorithms.

In order to keep running graph analytics and simulations with the best conditions, we propose adaptive parameter adjustment methods from the environment such as traffic status and current graph structures. In a traffic simulation, we add a system to our simulation framework to adjust the synchronization interval according to the congestion of whole road

network.

1.4 Contributions

With the proposal mentioned above, we summarize the contributions below.

1.4.1 Adaptive data assignments and parameter adjustment in an agent-based traffic simulation

First, we evaluated the performance bottleneck of our traffic simulation framework and application in a parallel and distributed environment, and found that the synchronization overheads and the waiting time caused by the load imbalances among all threads and processes is the primary problem of the parallel execution. Based on the finding, we proposed an adaptive and dynamic agent data assignment method to distribute the workload equally and synchronization frequency parameter adjustment method to reduce the synchronization overhead. Then, we evaluated these methods with the road network and commuter trip data in Dublin city, and showed our agent re-assignment method reduced the performance overheads for synchronization.

Here is the summary of our contributions:

- We developed parallel and distributed simulation framework named **XAXIS** to run existent an agent-based traffic simulation named **Megaffic** with multiple Java processes and threads.
- We evaluated the XAXIS with Megaffic using real traffic system data sets and generated trip data based on the public traffic census data, and state the problem of the XAXIS.
- We proposed performance optimization method to resolve the problems, and showed the reduction of the synchronization overheads from load imbalance.

1.4.2 Incremental community detection

First, we propose an incremental community detection algorithm based on overlapping and bottom-up community detection algorithm named DEMON [10] by extending these core functions to incremental fashion. Next, we propose a performance optimization method to one

of the DEMON core function to merge small community into larger communities, which is the main performance bottleneck of DEMON algorithm. Finally, we evaluated the performance of our incremental algorithm and compared the speed-up against the original DEMON algorithm.

Here is the summary of our contributions:

- We proposed an incremental community detection method as an extension of an existing community detection algorithm named **DEMON**.
- We also proposed an optimization method to avoid unnecessary small community merging operations in order to reduce performance bottleneck.
- We evaluated the efficiency of our proposed algorithm with time-evolving graph data set.

1.4.3 Adaptive and incremental graph pattern matching

In order to propose a proper incremental graph pattern matching algorithm for real-world graphs and applications, we defined the requirements (subgraph isomorphism, vertex/edge attributes and approximate patterns) for our graph pattern matching, and then we adopt an approximate subgraph isomorphism algorithm named G-Ray [6] as a base algorithm of our incremental algorithm. Next, we propose an incremental graph pattern matching algorithm by extending the core functions of G-Ray. In order to adjust the number of vertices for re-computations, we also propose an adaptive execution manager consists of reinforcement learning and graph clustering components. Then, we evaluated our adaptive and incremental graph pattern matching algorithm with time-evolving social network data. We compared the speed-up between our incremental algorithm and the original G-Ray algorithm, and the adaptive algorithm with our proposed execution manager and the naive incremental algorithm with the fixed parameters.

Here is the summary of our contributions:

- We proposed an incremental and approximate graph pattern matching method based on an approximate subgraph isomorphism algorithm named **G-Ray**.
- We also proposed an adaptive execution manager with reinforcement learning to select vertices for re-computation of the incremental graph pattern matching.

- To consider the efficiency of our incremental graph pattern matching, we gave quantitative evaluation of it with several types of data graphs and query pattern graphs.
- We evaluated the efficiency incremental graph pattern matching algorithm and the execution manager with real-world graph data sets and query patterns. Then we showed our algorithm gained speed-up compared to the baseline algorithm and the execution manager accelerated our incremental algorithm.

1.5 Thesis Outline

The rest of the thesis is organized as follows.

Chapter 2: Background and Related Work

In the chapter 2, we explain the overview of basic large scale graph analytic algorithms and agent-based simulation with real-world data set in order to describe the background and common problems more clearly. We also explain the state-of-the-art algorithms of community detection (and graph clustering) and graph pattern matching and agent-based related to our proposed methods.

Chapter 3: Adaptive Reassignment of Computational Resources and Agent Objects

In the chapter 3, we propose performance optimization methods of an agent-based traffic simulation in a parallel-distributed computation environment. In order to reduce the computation overheads from load imbalances and synchronization costs, we proposed inner-node threads assignments to agent objects, across-node threads and process assignments to agent objects, and adaptive synchronization interval. We evaluated combinations of these methods, and showed the best methods and parameters in a given data set and scenario.

Chapter 4: Incremental Community Detection

In the chapter 4, we propose an incremental community detection method named **IncrementalDEMON**. It is based on an existent bottom-up and overlapping community detection algorithm named DEMON [10] by an incremental extension for each subroutine of DEMON.

We also optimize a subroutine merging small communities to larger ones by judging whether the merge process is really necessary or not every time. We implemented the incremental extensions and optimizations and compared the efficiency of the incremental algorithm and the optimization with the real social network data.

Chapter 5: Incremental Graph Pattern Matching

In the chapter 5, we propose an adaptive and incremental graph pattern matching algorithm named **IGPM-PEM**. IGPM-PEM consists of incremental graph pattern matching (IGPM) component as an incremental extension of existent algorithm of approximate subgraph isomorphism algorithm named G-Ray [6] and adaptive subgraph chooser named partial execution manager (PEM) for re-computations of the IGPM. We evaluated the efficiency of IGPM and PEM respectively with various time-evolving graph data sets and query patterns.

Chapter 6: Overall Discussion

We state overall discussions related to our proposed methods in the chapter 6. We make algorithm-specific discussions about the validity of the adaptive optimization of Megaffic on XAXIS, DEMON-based incremental community detection algorithm and G-Ray-based adaptive and incremental graph pattern matching at the chapter 3, 4, 5 respectively. However, we did not yet verify the applicability or effectiveness of our optimization methods for general graph algorithms or simulations.

In this chapter, we discuss the common consideration applicable to other graph algorithms and agent-based simulation applications and frameworks. First, we discuss what kind of graph algorithms can we apply our optimization methods such as incremental and adaptive processing to. Second, we focus on various types of real-world graph data sets, parameters and scenarios. For example, some IGPM cannot gain efficiency with some kind of conditions including the shape of data graph and patterns. We discuss the applicability and limitations of our proposed methods.

Chapter 7: Conclusion

Finally we describe the conclusion of the thesis in the chapter 7. We also describe the directions of future research including generalization of our proposed methods, further optimizations to handle larger data.

Chapter 2

Background and Related Work

2.1 Real-world Applications with Graphs

Graph analysis is the core technique, and it is used for various fields of real-world applications such as social data analysis, bioinformatics, finance fraud detections, and cyber securities. In this section, we introduce these various real-world applications with graph analytics especially community detection, graph pattern matching, and agent-based simulations related to graph algorithms.

These graphs usually have millions of vertices and edges with various type of attributes (labels and properties). Moreover, some graphs are time-evolving, which means the vertices and edges are added and deleted, or these attributes are updated frequently.

2.1.1 Social Networks

In the last several years, social network services have been developing rapidly, and the number of users became more millions and billions, and the number of relationships between users (friendships and follower-following) is now hundreds of billions.

2.1.2 World Wide Web

Recently the number of web pages and hyperlinks connecting them in the World Wide Web (WWW) are increasing. Barabási et, al discovered the topology of the WWW network and its growth [11]. Its topology has a characteristics of scale-free as well as that of random

networks. That means the incoming and outgoing link distributions follow a power-law with random graph features.

In cyber security, graph-based outlier detection algorithms are applied for fraud detection [12] with log-in network data. Both community detection and subgraph detection (graph pattern matching) for dynamic graphs are used to find anomaly vertices, edges or subgraphs [13].

2.1.3 Finance

Graph analytics are also widely used in the finance fields. Many banks are trying to suspicious bank accounts, and transaction behaviors lead to money laundering or non-performing loans from billions and trillions of transactions among millions of accounts. In order to extract suspicious account groups and transaction sets, many graph algorithms are useful.

For example, outlier detection algorithms are used for fraud detection in suspicious amount or frequency of transactions between bank accounts. Many algorithms to detect such fraud transaction edges or subgraphs are also proposed, and some of them adopt a similar approach to find subgraphs recently updated frequently in cyber securities [13].

2.1.4 Transportation

In public transportation systems, road networks and route maps can be treated as large graphs with many attributes. Cross points, endpoints and other junctions are represented as vertices, and roads connecting junctions are edges.

Table 2.1 represents the statistical characteristics of road networks (the former three data sets) and social networks (the latter three data sets). Road networks usually have high diameters, and the degree distribution is different from other social networks and WWW hyperlink networks.

2.1.5 Bioinformatics

In a biological research field, graph analytics are also used for many applications. For example, community detection or graph clustering algorithms can be applied to protein-protein interaction (PPI) network to extract a group of proteins with similar functionality. Link prediction method can also be applied to PPI network to predict some undiscovered interaction

Name	Vertices	Edges	Average Degree	Diameter
Road-CA	1,957,027	2,760,388	2.82	865
Road-PA	1,087,562	1,541,514	2.83	794
Road-TX	1,351,137	1,879,201	2.78	1064
Flickr	404,733	2,110,078	10.43	18
LinkedIn	6,946,668	30,507,070	8.78	23
LiveJournal01	3,766,521	30,629,297	16.26	23

Table 2.1: Statistical characteristics of road networks (the upper three data sets) and social networks (the lower three data sets) from [14]. Many vertices (e.g. cross points) in road networks have smaller degrees while social networks have hub vertices with more degrees.

pairs.

Parthasarathy et, al. [15] showed that the subgraph mining and community detection algorithms are frequently used for classifications of biological networks. For example, both frequent subgraph mining (FSM) and community detection algorithms can be used to extract new protein functions from large PPI networks, and tree mining can be applied for RNA secondary structure networks to characterize RNA structure groups.

Zhang et, al. proposed GADDI [16], a distance index based subgraph matching especially for biological networks such as PPI networks and gene regulatory networks.

2.2 Characteristics of Real-world Graph Data

Recently, the real-world graph data, especially WWW, finance and human-interactions such as social networks becomes larger rapidly.

2.2.1 Dynamic Graphs

Most of real-world networks are time-evolving, and change the structure drastically.

Barabási [17] mentioned the growth of real-world networks such as WWW and proposed a new graph growth model considering such networks have characteristics of power-law as well as random graphs. The rapid growth of the WWW network became remarkable in 1999, and an old network growth model cannot be applied to these networks.

2.2.2 Large-Scale Graph Structures

The graph structures in the real world vary with applications. For example, social networks are usually scale-free, which of degree distributions follow a power-law. On the other hand, the road networks in traffic applications are near to mesh because most of cross points are connected to a few number of roads.

2.2.3 Applications of Dynamic Graphs

Time-evolving real-world data analytics with dynamic graphs are common applications with many research fields such as finance, cyber security, e-commerce, biology, etc.

2.3 Community Detection

Community detection is a graph processing to put the "close" vertices together into some groups or subgraphs. The metrics of closeness are different according to the algorithms, but it mostly defined with topological closeness. Community detection is one of the mostly used graph analytics for various real-world applications, and many algorithms have been proposed.

Graph clustering (partitioning) algorithms also divide a large graph into small subgraphs. Strictly speaking, community detection and graph clustering are different algorithm in that some community detection algorithms allow overlapping subgraph outputs and leftover vertices which do not belong to any communities [18]. In this thesis, we consider graph clustering algorithms are part of community detection algorithms.

2.3.1 Bottom-up and Top-down Approach

Community detection algorithms can be roughly categorized into "bottom-up" and "top-down" algorithms. In the "bottom-up" approach, it merges similar vertices or small clusters into large communities. In the "top-down" approach, on the other hand, it divides the original input graph or large clusters into small communities.

Louvain method: Blondel et, al proposed a graph clustering method called Louvain algorithm [19] to extract communities to optimize the modularity [20] of all communities. The definition of modularity Q is as follows [21]:

$$Q = \frac{1}{2m} \sum_{ij} \left(A_{ij} - \frac{k_i k_j}{2m} \right) \delta_{ij}$$

Note that m is the number of edges, A is the adjacency matrix, k_i is the degree of the vertex i and $\delta_i j$ is the Kronecker δ-symbol (if the vertices i and j is the same community, the value is 1, otherwise 0).

The Louvain algorithm is one of modularity maximization approaches. Because modularity maximization problem is NP-complete [22], the authors an approximate algorithm to iterate "modularity optimization" and "community aggregation" phases in order to handle larger graphs. The Louvain algorithm is relatively fast for million- and billion-scale graphs like the WWW networks, but the output communities are usually unbalanced.

Many optimized algorithms based on Louvain method are proposed for larger graphs. Dinh and Thai proposed polynomial-time approximation algorithms [23] for scale-free networks by modularity maximization. Traag proposed a faster Louvain algorithm by moving vertices to the community of a random neighbor while optimizing modularity [24].

Label propagation: Label propagation [25] is one of the main approaches for community detection. It assigns the unique labels to all vertices (initialization), and then each vertex updates the label of neighbor vertices with the highest frequency (label propagation). It iterates the label propagation phase, and the correctness of the result increases if the number of iterations is large. This algorithm is relatively simple and the computation complexity is only $O(m + n)$ when n and m are the number of vertices and edges respectively. The computation time of label propagation depends on the number of iterations, but only a few number of iterations is required to classify most of vertices correctly.

METIS: Karypis et, al proposed a graph partitioning algorithm named METIS [26], which uses a coarsing heuristic named "heavy-edge heuristic". The process of METIS consists of three phases; coarsening data graph into smaller graphs, partitioning of coarsed graph and uncoarsening. It can partition an input graph into small clusters with minimum edge-cuts, and tries to balance the output cluster sizes.

K-means clustering: K-means clustering is used not only for graph data but also for other general high-dimensional data. Unlikely to other community detection and graph clustering algorithms, it is the "top-down" approach, which repeats to divide the large graph into smaller clusters.

Spectral clustering: Many graph community detection and clustering algorithms use

bottom-up approaches with its topology. On the other hand, spectral clustering algorithm [27, 28] uses linear algebra computations to extract eigenvectors of the graph Laplacian. The elements of each index indicates the feature value of each vertex, and then it apply clustering algorithm for feature vectors such as k-means to classify vertices.

2.3.2 Overlapping Community Detection

Some community detection algorithms allow overlapped community outputs.

Gregory proposed an algorithm to find overlapping communities with a label propagation approach [29]. In this algorithm, each vertex has a unique label. When each vertex propagates its label to the neighbors, it sends its own label to neighbors with weights, and then updates the label from the received different labels with weights. Finally, some vertices have multiple labels with the same weights, that means these vertices belong to multiple communities.

Coscia et, al proposed an overlapping community detection algorithm named DEMON [10], which also partially uses label propagation. At first it create small subgraphs for each vertices named `EgoMinusEgo` network, then partition them into smaller clusters with label propagations, and then compare and merge all of these clusters into several overlapping communities.

Particularly, some community detection algorithms have been proposed for social network analysis [18]. Social networks usually have both local and global-scaled communities. Moreover, each vertex as a person have many attributes such as age, gender and interests. These attributes can be keys of more accurate community detections.

2.3.3 Incremental Community Detection for Dynamic Graphs

With the increasing demand of real-time community detection algorithms for time-evolving graphs, many incremental community detection algorithms including the extensions of existing methods have been proposed.

Zakrzewska and Bader proposed a dynamic community detection algorithms [30] named **BasicDyn** and its optimized version of **FastDyn** based on a greedy and modularity maximization approach. These algorithms store history of edge update histories, and merge them to detect the community changes.

Liu et, al. proposed a generalized incremental community detection framework [31] applicable to some bottom-up community detection algorithms such as Louvain [19], Label

Propagation [29] and DEMON [10].

2.4 Graph Pattern Matching

Graph pattern matching algorithms can be categorized into two groups based on whether it considers graph topology matching as well as vertex matching. One group is called "subgraph isomorphism", which also considers topologies between vertex pairs in the query pattern graph.

2.4.1 Subgraph Isomorphism

Subgraph isomorphism is one of the typical graph pattern matching problem. The definition of subgraph isomorphism is the following;

A subgraph G_s of the input data graph G matches the query pattern graph Q if there exists a (bijective) mapping function f from the vertices of Q to the vertices in G_s ($\forall u \in Q$ such that $f(u) \in G_s$) such that

1. For each vertices $u \in Q$ and $f(u) \in G_s$ has the same label.
2. There exists an edge $(u, u') \in Q$ if and only if there exists an edge $(f(u), f(u')) \in G_s$.

The subgraph isomorphism requires topological restrictions, but the computation complexity of subgraph isomorphism algorithms is NP-complete, so many algorithms of optimized subgraph isomorphism have been proposed such as Ullmann's method [32].

2.4.2 Graph Simulation

Graph simulation **graph simulation** [33, 34] is more relaxed graph pattern matching problems compared to the subgraph isomorphism. The definition of the graph simulation is following;

A graph G matches a pattern Q via *graph simulation* if there exists a binary relation $S \in V_Q \times V$, where V_Q and V are the set of the vertices in Q and G such that

1. For each vertex pair $(u, v) \in S$, u and v have the same label.
2. For each vertex $u \in Q$, there exists an edge $(v, v') \in G$ such that

(a) $(u, v) \in S$

(b) For each edge $(u, u') \in Q$, there exists an edge $(v, v') \in G$ such that $(u', v') \in S$

The computation complexity graph simulation is quadratic time, which is much less than subgraph isomorphism of which complexity is NP-complete. However, graph simulation does not guarantee the topological matching. Figure 2.1 shows an example of a matched pattern in graph simulation. However, the subgraph does not satisfy the requirements of subgraph isomorphism.

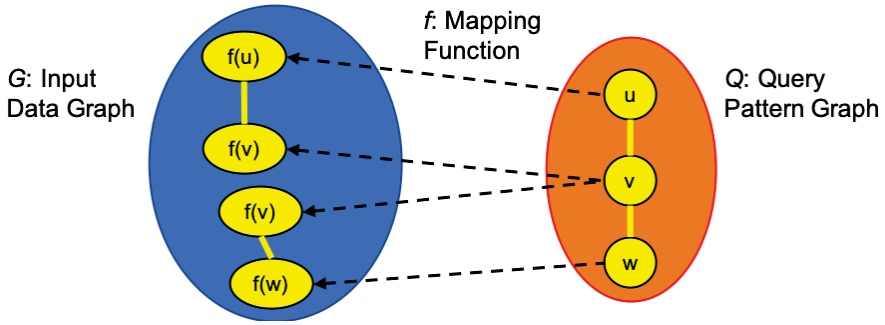


Figure 2.1: An example of a query pattern graph (right) and a matched subgraph in a input data graph (left) by graph simulation, but not subgraph isomorphism

Even the computation cost is smaller compare to subgraph isomorphism, it is still unreasonable for large-scale input data graphs. To reduce the computation complexity, other graph pattern matching problems based on the graph simulation such as **bounded simulation** [35] and **strong simulation** [36] are proposed. These graph pattern matching problems are based on the graph simulation, but they have some additional restrictions regard to the topological connections. In [37], Fan mentioned the computation complexity of the bounded simulation and strong simulation is cubic time while that of graph simulation is quadratic time.

2.4.3 Incremental Graph Pattern Matching

Incremental graph pattern matching algorithms are fundamental to handle time-evolving graphs efficiently.

Kao et, al. proposed a vertex-centric incremental graph pattern matching algorithm for streaming graph updates [38] working on distributed graph computing framework. The algorithm updates. The algorithm consists of three phases; update, self-checking and propagation.

When an edge is added or removed, the source vertex and destination vertex send the messages about the update and the matched pattern set. Next, each vertex checks the status of matching and send messages to itself. Finally, propagate the matching status to the neighbor vertices. Compared to the batch pattern matching algorithm, the incremental algorithm achieved from 3 to 10 times speed-up.

Fan et, al proposed incremental algorithms [39, 37, 40] based on graph simulation algorithms.

2.5 Agent-Based Traffic Simulation

Traffic simulation is a fundamental field of the social application. It is mainly used for city planning such as constructing new roads, road closures in irregular events and disaster preventions. In order to find the best traffic planning, it is important to simulate a number of scenarios with complex hyper-parameters. Moreover, the simulation area should be larger (e.g. city- or country-level) and the simulation time span (or the number of simulation steps) should be longer for more accurate simulation results.

Many traffic simulation applications use agent-based models. In agent-based simulations, each agent object has its states (e.g. positions) and a decision-making model. In agent-based traffic simulations, generally, each vehicle or pedestrian corresponds to an agent object, which checks the front vehicles, pedestrians and signals, and updates the speed, position and route for every simulation steps.

2.5.1 Motivation of the Parallel and Distributed Traffic Simulations

In order to gain global statistical insights, urban or country-scale traffic simulations must be conducted simultaneously because even a regional traffic congestion may affect the traffic status in remote regions. However, it is difficult to put all geographical data such as cross points and roads and all vehicle agents to a single computation node due to the limited memory capacity and number of CPU cores.

Consequently, a parallel and distributed execution with multiple nodes is indispensable. It is possible to port the simulation implementation to C/C++ to reduce the memory overhead and gain the efficiency of the CPU usage, but distributed simulations are fundamental for certain scale of road networks because the use distributed computation environments reduce the memory usage for each computation node and handles more CPU cores.

2.5.2 Simulation Framework for Parallel and Distributed Environments

Some frameworks support parallel executions in distributed environments. However, agent assignments for agent-based social simulations is a critical problem especially in parallel and distributed computation environments to avoid load unbalancing and keep scalability with the number of threads and processes.

D-MASON [41, 42] is a parallel agent simulator in distributed environments extended from MASON [43]. MASON is a multi-agent simulation framework. All agents in MASON run step by step, but each agent runs in its own thread asynchronously within a step.

dSUMO [44] is a traffic simulation in distributed computation nodes. dSUMO is developed as an extension of an agent-based traffic simulation framework named SUMO [45]. In order to communicate other physical machines which the same simulation instances are running on, the authors defined *dSUMO Container*. Each machine has a container to manage the SUMO processes in this machine and communicate with other containers in the different machines.

Both of D-MASON and dSUMO are supposed to use multiple remote servers including commodity desktop computers.

2.5.3 Traffic Simulation and Graph Algorithms

In traffic simulations, many graph algorithms are used for preprossessing and postprocessing. For example, shortest path extraction algorithms are frequently used to compute routes. When a traffic simulation is executed in a parallel and distributed computation environments, we need to partition road networks into small clusters.

Shortest Path Extraction

All agents in traffic simulations (vehicles and pedestrians) need to compute the route from the origin and destination points and their preferences (minimize travel times, distances or tolls) in advance. In traffic simulations, many shortest path computation methods [46, 47] based on Dijkstra algorithm have been proposed.

Graph Partitioning

As we mentioned in the beginning of the section, large-scale and long time-span traffic simulations with many scenarios are important to output accurate results. Parallel and distributed

computation environments and a better graph partitioning algorithm are indispensable for such large-scale traffic simulation in the realistic time by distributing road network subgraphs into computation processes. In agent-based simulations, it should minimize the synchronization and agent migrations to keep scalability. The requirements of a graph clustering algorithm for agent-based traffic simulation are as follows:

Balanced subgraphs Generally, the simulation runtime for each iteration in a partition of road network depends on the number of vehicle agent objects running in the partition, and these numbers for all partitions should be always balanced. In this case, using graph partitioning algorithms which accept vertex-weighted graphs is better to balance the total workloads for each thread and/or process and to reduce the idle time.

Minimum edge-cut Communication cost as well as load balancing is critical for parallel and distributed executions. In agent-based traffic simulations, when some vehicle agents running on a partitioned road network are about to move to another road in the different partition, these agent objects needs migrations from a process handling the current road network to another process handling the destination road network. The cost of these migrations is remarkable because it contains costs of synchronizations between these partitions, serializations and deserializaitons of agent objects, and communication overheads. In order to reduce the costs, we have to minimize the frequency of the migrations, and we have to partition the original road network with minimum edge-cut to reduce the frequency.

Small computation complexity It is ideal that the number of running agents in each partition are totally equal and the total edge-cut is minimum, but the computation complexity of exact graph partitioning algorithms is too large to execute for every times. Therefore, approximate algorithms with low complexity is necessary not to affect the performance of simulation executions.

In chapter 3, we partitioned road network using METIS [26] with the minimum edge-cut option to reduce the frequency of migration. Recently we conducted traffic simulation in Tokyo bay are network [48]. Figure 2.2 shows the partitioned Tokyo bay area network into 16 subgraphs.

In dSUMO [44], the authors tried some road network partitioning methods; uniform and irregular square partitioning, graph partitioning and SParTSim [49]. SParTSim partitions road networks based on the hierarchy of roads (e.g. expressways or regional roads) to minimize inter partition communication in SUMO.

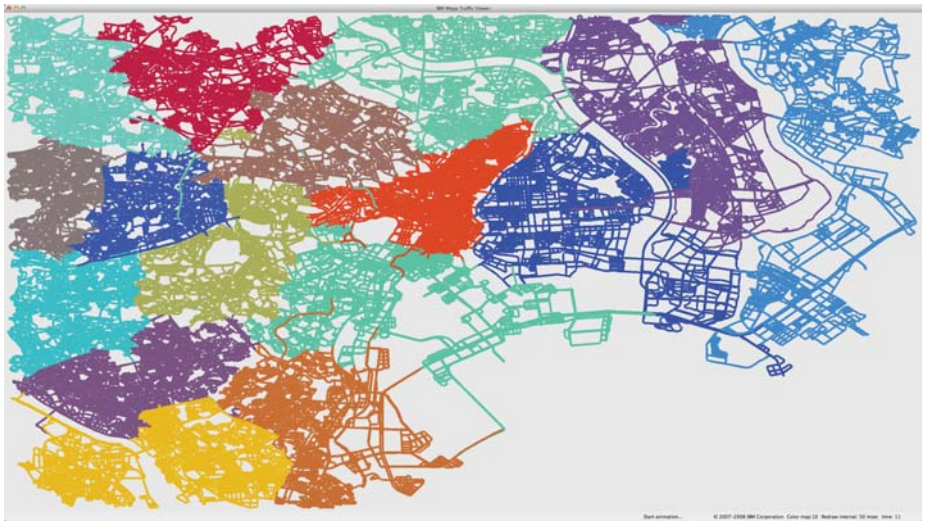


Figure 2.2: Road network of Tokyo bay area retrieved from [48], which is partitioned into 16 subgraphs to minimize edge-cut.

Chapter 3

Adaptive Assignment of Agent Objects

It is indispensable to make full use of parallel and distributed systems with increasing demands for large-scale traffic simulation, but problems remain about insufficient scalability due to costs of synchronization by load unbalancing among compute nodes. To tackle this problem, we propose performance optimization methods for traffic simulations to underlying road networks prepossessed by graph contraction introducing dynamic re-assignment vehicles and cross points to threads and nodes based on time-series traffic congestion. By applying the optimization and running the simulation of the real-world Dublin city on 16 compute nodes of TSUBAME 2.5, the simulation performance has improved by four times with the proposed graph contraction method. We compared the effect of optimizations between the agent assignment method and the existing adaptive synchronization method with the comparison to regular one synchronization per step.

3.1 Introduction and Motivation

In recent years, demands of large-scale traffic simulation are increasing from disaster prevention and city plannings. Such simulations are required to be faster especially about evacuation simulation against sudden disasters. With increasing demands for more massive simulations, it is hard for a standalone computer to simulate large-scale social behavior in that all agent objects exceeds memory capacity. Even a large single computation node can handle large road networks with millions of cross points and connecting roads, it takes more than several

minutes to simulate single simulation step corresponding one second in a real-world traffic. Therefore, large-scale road networks must be partitioned and distributed to many computational threads, processes and nodes. Figure 3.1 is the visualization of a traffic simulation in Japan.

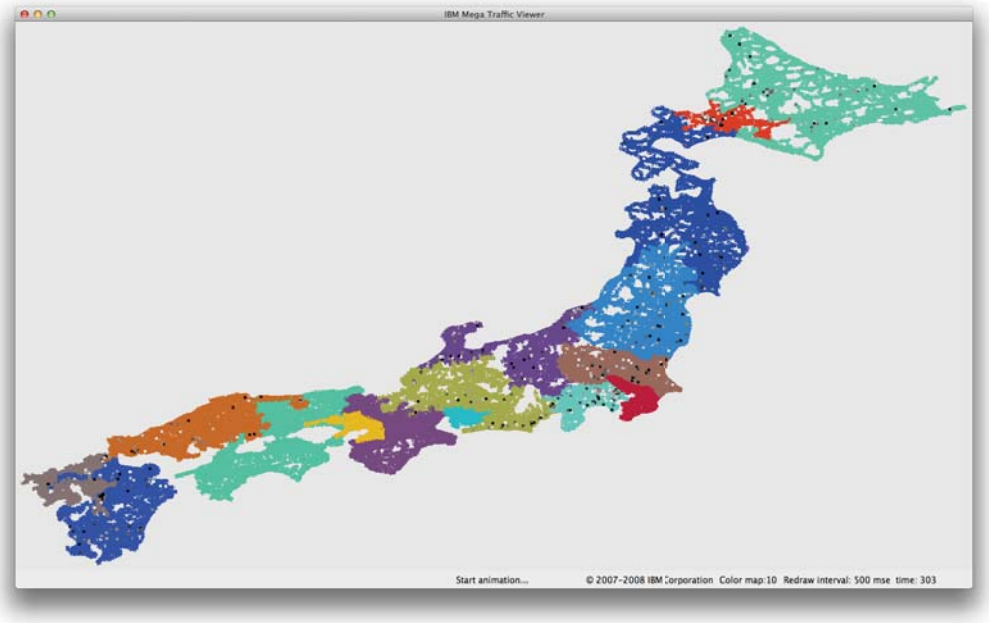


Figure 3.1: The traffic simulation in Japan with a million cross points, two million connecting roads and 10 million vehicles. Because it takes more than a few minutes per simulation step corresponds to one second in the real world in a standalone computation nodes, we partitioned the road network into 16 partitions and distributed them to Java processes.

Consequently, parallel distributed computation environments are usually used to handle large-scale simulation objects in a shorter time by distributing these objects to each computation nodes.

As a means to apply an agent-based simulation program to distributed computation nodes, however, there are some problems related to the performance we have to overcome. First, distributions of agents and other objects must be distributed equally to each computation nodes for better speed-up and scalability. Unfortunately, such distribution will change dynamically in traffic and other ITS simulations according to as running simulations. For a reason, the scalability will be sometimes worse while conducting such simulations for a long time. Second, communication cost between CPU cores and computation nodes must be taken

into consideration. Agent objects will have to be distributed to each node in order to they can handle in-memory, but some agents will send data to other neighboring agents at the different nodes. The effect will be more significant by increasing the number of nodes and processors. Moreover, all nodes must synchronize with each other frequently to keep consistency. When some nodes simulate their role area very faster, and the other do slower, inconsistent outcomes may be generated.

In this research, we executed agent-based traffic simulation in a distributed parallel simulation base and showed that the principal overheads are load imbalance between nodes and synchronization time. Then we proposed and applied a method to counter these overheads to our simulation base, and evaluated effects of the method. We also discussed the most suitable scale of computation environments with input data.

Our contributions are following.

- We found the primary overhead of agent-based traffic simulation in parallel distributed environment.
- We proposed more effective method for agent-based simulation and implemented it to our agent simulation platform.
- We evaluated the method in several clusters up to 16 computation nodes and found limitations of speed-up by the number of nodes.

The rest of the paper is organized as follows. Section 2 describes our agent-based simulation platform for the parallel distributed environment. Section 3 describes an agent traffic simulator we used. In section 4, we explain problem findings of performance overhead. In section 5, we present our proposed method. In section 6, we present evaluation conditions and results of our implementations. Section 7 describes related work. In section 8, we conclude and show future work.

3.2 XAXIS: Agent-based Simulation Platform

As previous work, we developed XAXIS (X10-based Agent eXecutive Infrastructure for Simulation)[50, 51, 52] as agent-based simulation middleware. Simulation applications run on each X10 places independently, and XAXIS supports multiple node execution like message exchanges, agent migrations and synchronizations for all places. XAXIS is based on ZASE[53] simulation base, which can run on a single node. We extended ZASE using X10 so that ZASE applications

can be executed distributed parallel computation system to handle larger environment and gain scalability. XAXIS provides the same API as ZASE, and the same applications can be executed the parallel and distributed environment without any modifications. Figure 3.2 represents the software stack of the agent-based simulation.

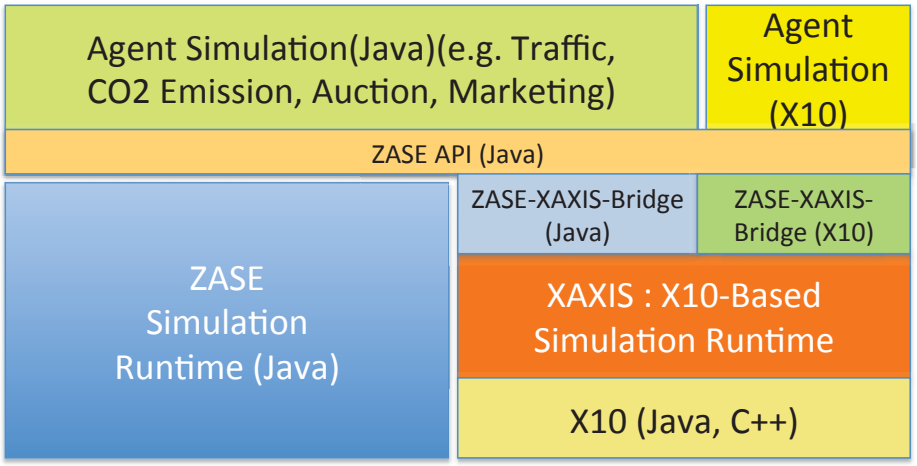


Figure 3.2: Agent-based simulation software stack with XAXIS, ZASE and Java-based simulation applications. The simulation applications can run at parallel and distributed applications with the same ZASE API.

XAXIS executes following procedures at a simulation step and repeats: handles agents, exchanges messages, and exchanges migrating agents. XAXIS entrust these simulation procedures with all X10 places, but in order to keep accuracy, it has some check points where places will stop until rest places reach this point in every simulation steps. Some places will wait until all places handling agents by X10 Team library so that they can send messages and agents to correct destination.

3.3 Megaffic: IBM Mega Traffic Simulator

Megaffic (IBM Mega Traffic Simulator)[7] is agent-based traffic simulation application. It can run in the distributed node system on the XAXIS simulation middleware. Megaffic used to run on ZASE API, so we improved methods about agent behavior a little.

In the Megaffic application, cross points handles incoming roads to move vehicles on the roads. Moving vehicle method is implemented at Java classes related to cross point, and

vehicle repositories is implemented at road class. When vehicles are about to run on the roads, destination cross point of the running road extract vehicles object from road and adjust speeds and positions of the vehicles one after another. If some vehicles reached to a cross point, the cross point calculates the next road and cross point, and then leave the vehicle with the cross point. In this simulation program, each cross point is correspond to Driver Agent at XAXIS, and vehicle is corresponded to Citizen Agent because the road network is always stable but vehicles will be handled different cross points in other places.

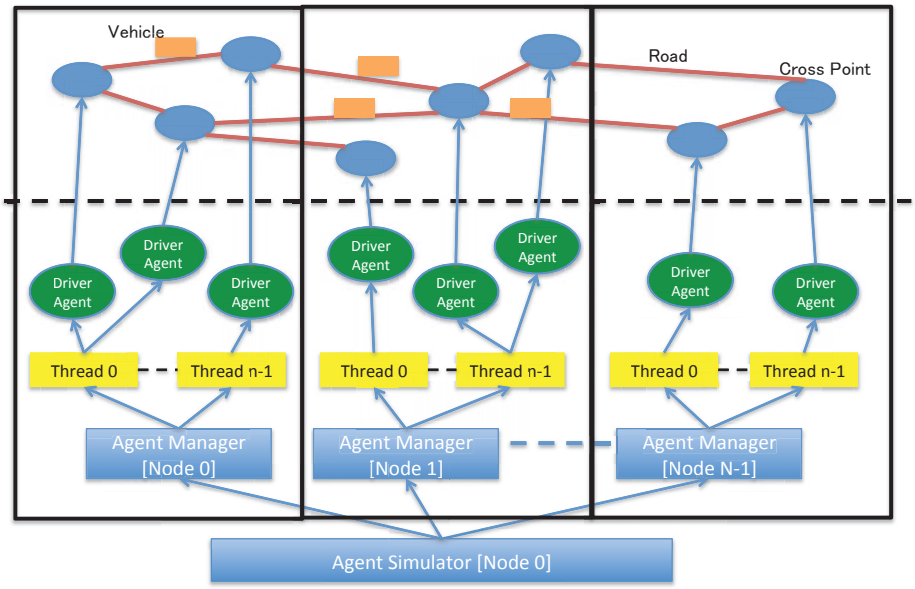


Figure 3.3: Agent assignment and handling from XAXIS to Megaffic. Each X10 place (process) invoke X10 activities (threads). These threads handle agent objects (cross points) and control vehicle objects via these cross points.

Figure 3.3 describes how XAXIS works and executes the Megaffic application in the parallel computation environment. In XAXIS simulation platform layer, Agent Simulator will be invoked at first when simulation program is launched. Agent Simulator creates Agent Manager at each computation node (X10 Place), and invoke them to run Driver Agent objects. These Agent Manager objects start several Java threads (X10 Activities) to handle Driver Agent objects. Each thread call run method at Driver Agent, and Driver Agent objects will make Megaffic application objects work.

In the Megaffic application layer, all cross points will call method to move vehicles running on inner roads. Actually the Megaffic class of vehicle is treated as subclass of Citizen Agent

on XAXIS and managed by Agent Manager in XAXIS, but vehicles will be moved by Driver Agent via cross points and roads.

In traffic simulation at multiple-node execution, the road network as simulation subject will be divided to sub-graphs to be stored each node. Some vehicles will be handled by a cross point in the other nodes when running Megaffic application at distributed environment. In this situation, the vehicle objects should be sent to the destination nodes. XAXIS supports Citizen Agent migration like message-passing methods. First, the application creates a message object and store the vehicle (subclass of Citizen Agent) to this message. The message is sent from application, and restored as vehicle objects.

3.4 Preliminary Evaluation and Problem Findings

The execution speed should increase as the program use more computation nodes because the number of agents (cross points) each thread or node handles will decrease. However, some problems will happen when agent-based traffic simulation program is executed in parallel distributed systems. Many vehicle objects run in a long distance, and sometimes they will have to reach other cross points in the different nodes. In this situation, vehicle object implemented as Java class must be serialized and deserialized. And all nodes must synchronize each other to keep consistency.

Furthermore, some specified nodes will have to handle many vehicles while the others will handle few of them because vehicles move to them in some simulation scenarios. Even all vehicles are distributed to all nodes equally, congested and clear areas will be separated with simulation running. It causes a lack of scalability because of workload imbalances between nodes and more wait time for synchronizations.

To detect the main performance overheads, we evaluated by preliminary execution with some computation nodes. The result of simulation is Figure 3.4. **MoveVehicles** means average elapsed time that each Java thread handles cross point and have vehicles to run. **SendMessages** means the elapsed time that each vehicle sends messages to neighbor vehicles. Migration means the elapsed time that all computation nodes send vehicle data to other nodes.

Our simulation program got the best performance at 6 nodes, and there are no performance improvements at more than 8 nodes. Procedure time of cross points' moving vehicles was decreased as increasing the number of nodes and the average time at 16 nodes is about 1/16 of 1 node execution. However, the time is calculated as average, and about 80% of

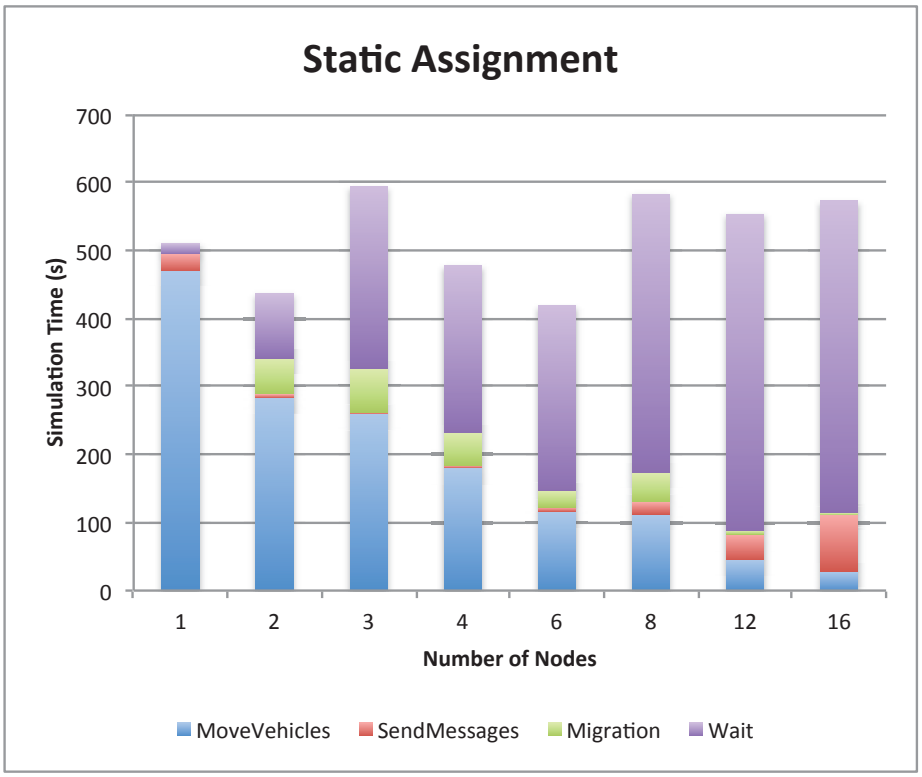


Figure 3.4: Simulation time details in preliminary execution. With increasing the number of computation nodes, the elapsed time to move vehicle decreased, but the waiting time for synchronizations drastically increased. In the result, the number of nodes with the least simulation time was 6.

total simulation time is wait time for finishes of other nodes. Communication overheads by migrating vehicles to other nodes are also remarkable in multiple node execution.

3.5 Optimization Methods

We propose several performance optimization methods: dynamic agent assignment inner nodes (to threads) and assignment across nodes (to threads and nodes). We compare effects of these methods to reducing synchronization frequency method.

3.5.1 Synchronization Interval Extension

At first, we implemented our existing idea [51] to XAXIS.

In the XAXIS simulation base, all computation threads and nodes have to synchronize each other in order to send and receive messages and agents correctly. In this system, however, most nodes will be idle while some nodes take a long time to manage many agents. And what is worse, when all threads and nodes synchronize, each node will have to communicate to the other nodes. Those result lack of scalability.

To reduce these synchronization overheads, we added frameworks to adjust constant synchronization interval to XAXIS. If the interval is set to longer, each node can execute independently and does not need to communicate except sending messages and agents asynchronously. Figure 3.5 shows the work flow of the optimization method.

3.5.2 Dynamic Agent Assignment

We found that the main overhead of agent-based simulation is load imbalances by distribution of agents.

To overcome the load imbalance problem, we propose dynamical Driver Agent assignment methods as one of the optimization method types. One is completed at each node, only reassign threads to agents. The other is more costly, but also more powerful method re-assigning agents among X10 Places.

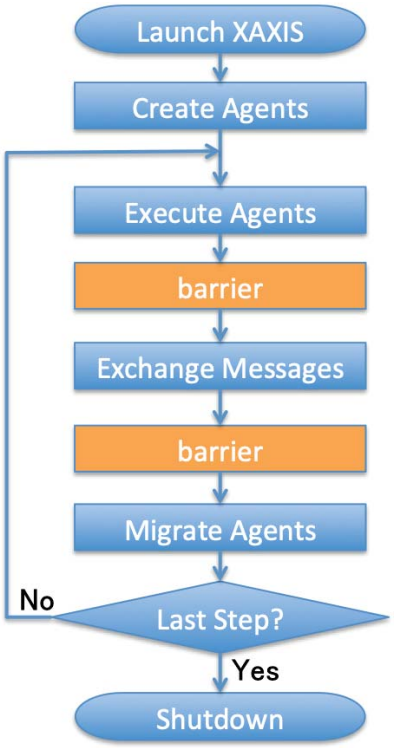


Figure 3.5: Synchronization interval adjustment work flow.

DYNAMIC AGENT ASSIGNMENT AT EACH INNER NODE

The first assignment method changes only the mapping between threads and Driver Agents. It is completed in each node, and no vehicles will be migrated during this assignment. In order that the workload will be equal among threads at each place, the mapping what thread will handle cross points in this place regularly. There are no change among road network and vehicles.

In the whole of a simulation, the number of cross points handled by each node is not changed. That means load imbalance among distributed nodes will not be resolved. However, the procedures of re-assignment are very simple and no extra vehicle migration is not necessary.

DYNAMIC AGENT ASSIGNMENT ACROSS NODES

The dynamic agent assignment inner nodes will keep load balance in each node, but load imbalance between nodes is not yet resolved. The reason is that the total number of vehicles in each node may change drastically. To tackle that problem, we also proposed and implemented agent assignment across nodes together with handled vehicles.

In this method, all nodes have whole road network. It will require more heap memory and extra runtime costs, but the usage of memory from cross points and roads is much less than that of vehicles.

The agent assignment procedure is shown at Figure 3.6.

1. When some cross points should move to other nodes, deactivate them (stop thread assignments) and no threads will handle such cross points.
2. In order to continue moving vehicles, migrate these vehicles to other nodes where these handling cross points will be activate.
3. Activate cross points (assign them to threads) in the destination nodes.

By re-assigning cross points with vehicles, all vehicles handled by the cross points need to migrate to destination node. It causes other performance overheads for communications and calculations. However, by the communication among all nodes, load imbalance will be resolved in the whole simulation environment.

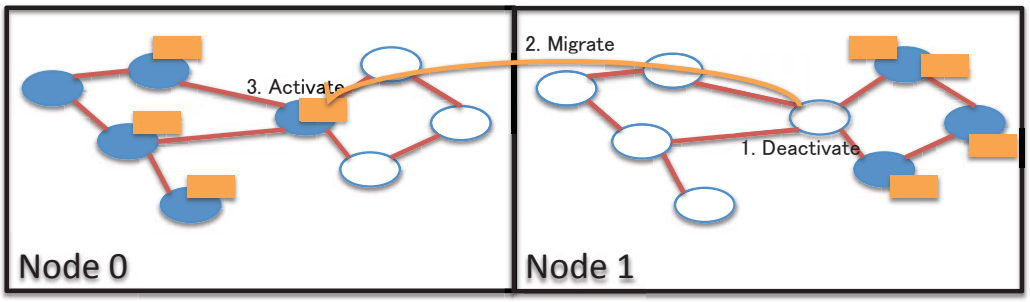


Figure 3.6: Agent assignment across nodes. First, deactivate the cross points of the origin of the migration. Second, serialize the vehicle objects for the migration and send to the destination cross points in the remote computation node. Then, deserialize the object to the destination cross points and activate these cross points.

3.6 Evaluation

3.6.1 Input Data Set

For preparing input files for traffic simulation, we obtained Open Street Map (OSM)[54] XML file and General Transit Feed Specification (GTFS)[55] files as road network data and person trip file as vehicle trip data. However, Megaffic application can only input specify formatted CSV file, so we had to convert these XML and GTFS text files to the CSV files.

Because curved roads in OSM road network data are represented as many connected short roads, it will be hard to handle whole network as it is with fewer computation nodes. In order to improve efficiency, we connected such roads to one straight one. In this optimization, curved roads are treated as straight ones. However, the affect of the optimization is little because road network structure will never change and the lengths of connected roads are sum of short straight ones. After the contraction, the number of cross points is reduced from 118,856 to 19,062 and the number of roads is from 269,869 to 40,965.

After aggregating networks from input data, we divided the road network into the number of computation nodes or threads. In default, XAXIS reads a definition mapping file what agents are assigned to each X10 place. We used METIS[9] as network partitioning program to generate a mapping file from agent ID to X10. In the default option, METIS divides network into the non-contiguous network. It has option to try to partition each network cluster contiguous, but this option can be applied only when input network is contiguous. We extracted largest Strongly Connected Component (SCC) as input data of METIS to apply

"contig" option, which means it tries to make output clusters are contiguous. Although the original OSM network has many isolated cross points and roads or very small road networks, we removed them because such networks are not related to traffic simulation as no vehicles will likely run on them.

For dynamic agent object re-assignments, we partitioned the snapshots of road networks from the previous simulation logs for each 1,800 steps in order to balance the workload of all computation nodes and threads. First, we create road networks with weighted vertices (cross points) by the number of vehicles running in this step. Then, we partition the road network with METIS so that each partition have the same sum of vertex weights (Figure 3.7).

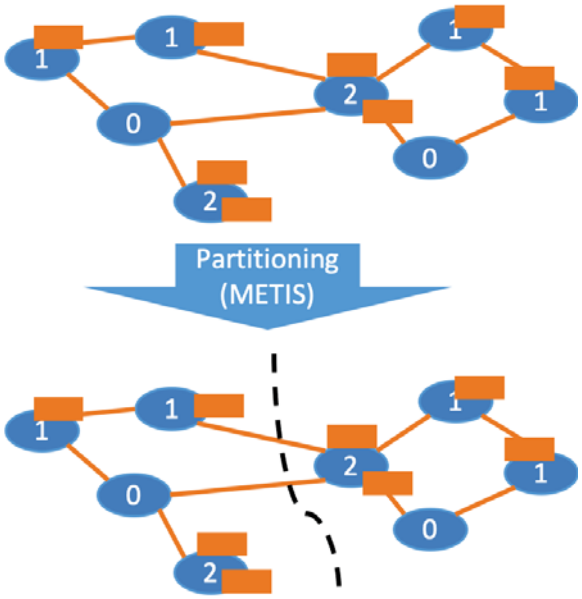


Figure 3.7: Partitioning the weighted network with METIS in order to all computation nodes and threads can handle the same number of vehicles.

We prepared 234,378 person trip data about commuting in Dublin city for vehicle routes. Most of trips will go from suburbs to Dublin City Centre. For privacy preserving, we obtained limited data: the location of origin, destination and departure time for each person. Megaffic application needs full path data including all cross points to be passed, so we have to generate trip paths for all people from our generated road network.

In the Megaffic application, one simulation step is corresponded to one second at default setting. In our experiments, simulation steps are set to 14,400, which is corresponded to 4 hours. The start time of trips is based 6:00 AM, so this experiment is expected people

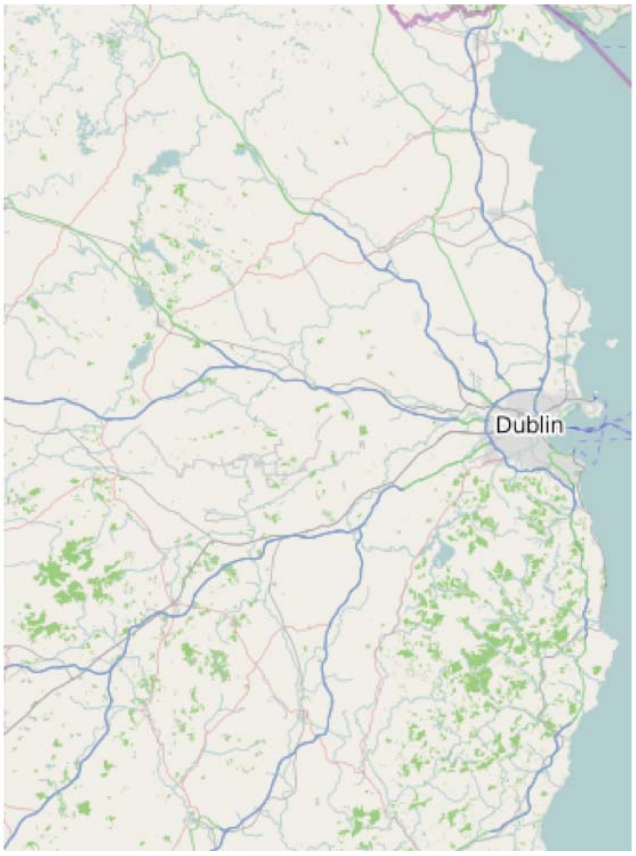


Figure 3.8: Road network of Dublin city and the suburbs from [54] (©OpenStreetMap Contributors). Many people commute from the suburbs to center of Dublin city using vehicles via expressways (blue lines).

commuting simulation from 6:00AM to 10:00AM.

3.6.2 Environment

We executed simulation program at TSUBAME 2.5 supercomputers Thin-nodes at Tokyo Institute of Technology . The specification data is Table 3.1. These Thin-nodes have GPU cores, but we used only CPU cores because of XAXIS architecture.

CPU	Intel Xeon X5670 2.93 GHz (6 cores) x 2 (Hyperthreading enabled)
Operating Systems	SUSE Linux Enterprise Server 11 SP3
Memory	PC3-10600 54GB
Network	QDR InfiniBand x 2 (80Gbps)

Table 3.1: TSUBAME 2.5 Thin-Node Specification

Megaffic application and XAXIS simulation base are running on Java and X10. The version of Java and X10 is following.

The parameters are shown at Table 3.2.

Number of X10 Places	1, 2, 3, 4, 6, 8, 12, 16
Number of X10 Threads	12
Maximum Java Heap Memory	48GB
Minimum Java Heap Memory	24GB
Java Stack Size	64MB
X10 Version	2.5.1
Java Version	java version "1.7.0_09" Java(TM) SE Runtime Environment Java HotSpot(TM) 64-Bit Server VM

Table 3.2: Multi-Node Execution Parameters

In X10, more than one places can be launched in a node. For example, if we launch XAXIS application with setting environment valuable X10_NPLACES to 2 at single node, two Java Virtual Machines will launch at the same node. But we assigned one place for each node in this time because we wanted to use sufficient heap memory.

3.6.3 Performance with Reduction of Synchronization Frequency

When we reduce synchronization frequency, the simulation time reduced at more than 4 nodes. Especially at 16 nodes it reduced by 35% while there was little effect at less than 3 nodes. However, the best number of nodes was 6 and the simulation time at more than 6 nodes became worse. The main reason is that the number of message objects including vehicle objects sent between physical computation nodes were increased and our method could not cover these message handling overheads.

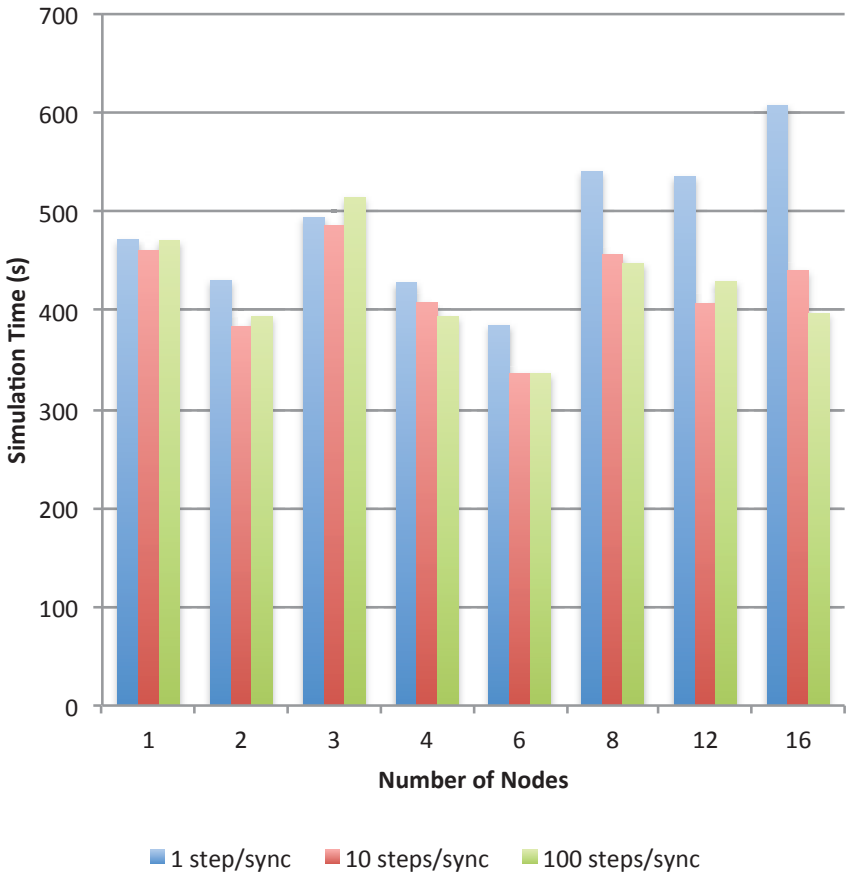


Figure 3.9: Simulation time by changing synchronization interval.

3.6.4 Performance with Dynamic Agent Assignment

We performed experiment with each our optimization method, dynamic agent assignments inner and among nodes. The result of reduction of dynamic agent assignment method is Figure 3.10.

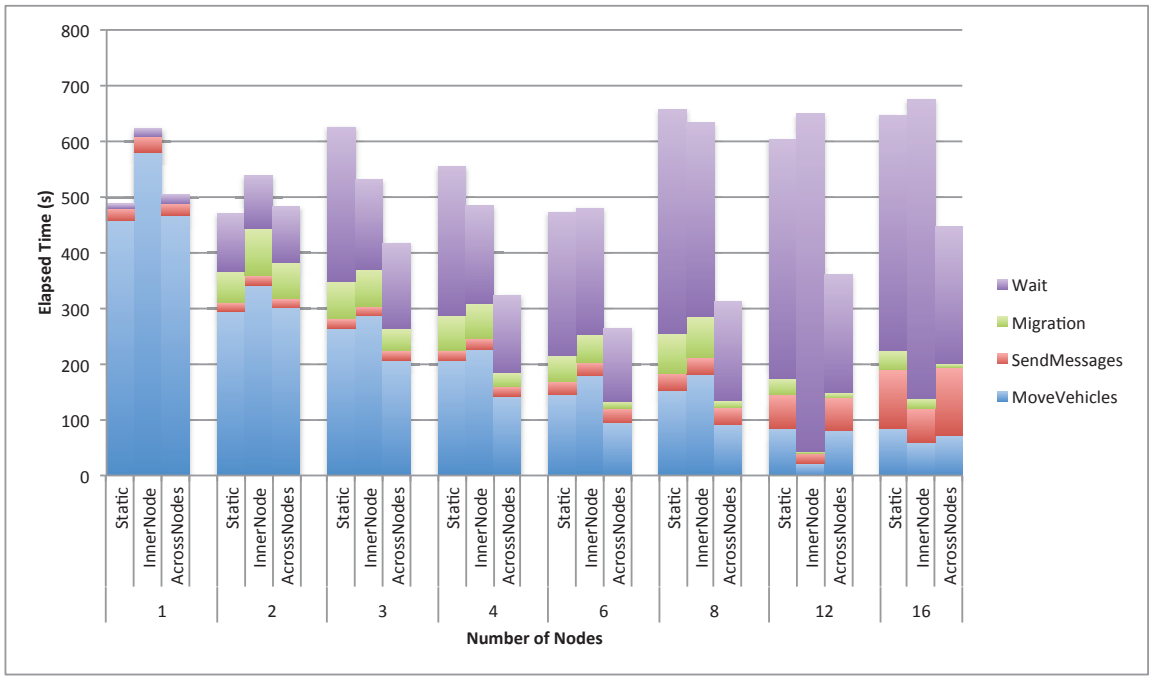


Figure 3.10: Simulation time with dynamic agent assignment method.

3.7 Related Work

As our previous work [51], we implemented dynamic adaptive synchronization method to XAXIS and Megaffic. The synchronization overheads are remarkable, so we added a framework to control synchronization according to congestion of road network. However, the origins and destinations for trip of vehicles are selected randomly, so load imbalances were hidden. In this our paper, we used real origin-destination data and found load imbalances as new kind of overhead.

MATSIM[56] is one of the most popular traffic simulator. It is on large-scale agent-based system, and many simulation models can be applied like mixture of public and private vehicles.

There are many related researches about MATSIM, most of which are case studies on many cities or model extensions. In [57], the authors tried to gain scalability of MATSIM by Java thread. They reached 6 times speed-up using 8 threads. But these evaluation simulations are performed standalone system, not distributed computation nodes.

SUMO[45] is also microscopic traffic simulator, implemented in C++. It is said that there is no limitation the volume of input data like network size and the number of vehicles, but not mentioned that it will work on distributed parallel system.

dSUMO[44] is an extended version SUMO of distributed feature, and it is evaluated using Dublin road network. The author implemented Container and interfaces such as Server and Clients. However, the authors used only primary roads, and evaluated with up to only 3080 vehicles, and they compared performance up to 2 machines and 4 CPU cores.

3.8 Summary

We will have to evaluate our simulation program with these optimization methods for larger road networks and more vehicles. In these experiments, we used whole Dublin city road network and found that the result is the best at 6-node and 72-core execution, and that performance was saturated and become worse at more than 6 nodes. The reason is that the problem size is fixed and these experiments are performed at strong scaling. Using too many computation nodes caused increase of communication costs like vehicle migrations. In order to show these optimization methods are effective, we need further experiments with various road network or trips.

Our optimization methods leave room for improvements. In our current implementation, whole agent assignment rules are decided before running simulation. The same preliminary simulation have to be executed and obtain log files about the number of running vehicles on roads and cross points. It causes waste of simulation time and computation resources, so we will have to re-assign agents automatically while running simulations by monitoring workloads at nodes or threads.

It also needs much more sophisticated improvement on XAXIS simulation base to synchronize only partial calculation nodes at our current simulation base, so we will re-design XAXIS to be available more elastic model as our future work.

Furthermore, it needs additional evaluations with different traffic data sets to verify our proposed methods. Traffic maps of various scales of cities are available from OSM, but it is

difficult to obtain public real vehicle trip data because of privacy issues. Our optimization method assumes real traffic situation with unbalanced road congestion, so we have to consider how we can generate trip data from limited traffic information and apply to our simulation program.

Chapter 4

Incremental Community Detection

Community detections (community discoveries) for large-scale real-world networks have been more popular in social analytics. In particular, dynamically growing network analyses become essential to find long-term trends and detect anomalies. In order to analyze such networks, we need to obtain many snapshots and apply the same analytic methods to them. However, it is inefficient to extract communities from these whole newly generated networks with little differences every time, and then it is impossible to follow the network growths in the real time. We proposed an incremental community detection algorithm for high-volume graph streams. It is based on the top of a well-known batch-oriented algorithm named DEMON [10]. We also evaluated performance and precision of our proposed incremental algorithm with large real-world networks with up to 410,236 vertices and 2,439,437 edges and computed in less than one second to detect communities incrementally - which achieves up to 101 times faster than the original algorithm without sacrificing accuracy.

4.1 Introduction of Community Detection

Community detections for large-scale networks have been significant research problems in graph computing, and real-time analyses of complex social networks are also essential to discover trends. Most of the real-world networks like user-relationships on social networking services are time-evolving and dynamic networks. Consequently, in order to detect and extract communities from large-scale and time-evolving dynamic social network in nearly real-time fashion, it is inefficient to compute them from the beginning of the entire network every time, even though there is little change. To overcome the problem, several approaches have been

proposed for community detection algorithms working in an incremental fashion, which is often called incremental community detection.

We propose an incremental community detection algorithm based on one of well-known and scalable community detection algorithms named DEMON [10] for incrementally growing networks. We implemented the incremental algorithm in C++ and conducted performance evaluations with real-world networks representing relations of people. Then we showed that our method took only less than a few seconds to compute incrementally updated networks while original DEMON needed to execute community detections for a whole network in a batch fashion - which brought the tremendous amount of overhead.

Our contributions are following: we proposed an incremental version of DEMON algorithm with DEMON devising core functions, reduced unnecessary community comparison procedures by defining a community mapping table, and proved the effectiveness of these proposed optimizations using real-world networks.

4.2 DEMON - An Overlapping Community Detection Algorithm

In this section we will introduce an overlapping community detection algorithm called DEMON, since our proposed incremental algorithm uses this as a base algorithm.

4.2.1 DEMON Overview

DEMON [10] is a scalable algorithm for detecting overlapping communities in complex networks. In this context, a community is a group of nodes densely connected to each other. It outputs overlapping communities, that means some nodes will be in different communities at the same time. We choose this algorithm as baseline of our research with several reasons.

First, DEMON has fine-grained scalability. Because communities will be generated in a bottom-up way and some of the required computation for all the vertices can be independently executed, it would fit with parallel and distribution computation. We considers it is also suitable for incremental extension.

Second, it is the state-of-the-art community detection algorithm that outperforms other algorithms in terms of accuracy. DEMON algorithm allows overlapped community outputs, that is usual for real-world networks. Most of large-scale social networks contains many

vertices with multiple properties, so DEMON algorithm should output proper communities from these properties. Moreover, The author of [10] showed DEMON provided higher accuracy than other community detection algorithms in that the F-Measure scores based on labels of vertices are the largest. They also described that the community size distribution of DEMON were more balanced than that of Infomap [58] algorithm.

Furthermore, three steps of DEMON algorithm including *Ego Minus Ego*, *Label Propagation* and *Merge* functions are performed for all vertices in an independent manner. Because each vertex is processed independently, some functions could be executed by incremental procedures.

4.2.2 Algorithm Details

DEMON algorithm repeats the following functions including *Extracting Ego Minus Ego Networks*, *Label Propagation* and *Merge* functions for each vertex in the input network.

1. Extract an *Ego Minus Ego* network from a vertex.
2. Group vertices from the extracted *Ego Minus Ego* network according to the result of label propagation.
3. Merge generated groups by the overlapping degree.

Next, we will describe details of these three functions DEMON algorithm consists of.

4.2.3 Extracting Ego Minus Ego Networks

In DEMON algorithm, the first step extracts an Ego Network for the chosen vertex. Ego Network is a sub-graph of original input network consists of the specified vertex (Ego), vertices neighboring the Ego vertex and edges connecting these vertices. Obviously a single node connecting the entire sub-graph are connected directly and affect the similarities of neighboring vertices, even if they are not in the same community. For this reason, the ego vertex is removed from its own ego network in this function. *Ego Minus Ego* network is a sub-graph removed the Ego vertex from the Ego Network.

4.2.4 Label Propagation

After the *Ego Minus Ego* graph is extracted from the specified vertex, the next step is to compute communities subsets of this network by *Label Propagation*. The label propagation method is based on [25], updating vertices labels by other label frequency of neighbor vertices.

This function repeats the following procedures to find small communities with the graph structure.

1. Set numerical labels to all vertices. Each label must be different from those of other vertices.
2. Set time stamp $t = 1$.
3. Choose a vertex from the network randomly and put labels gather from neighboring vertices.
4. Find the majority of gathered labels and set it as the new label of this vertex.
5. After processing all vertices, if all labels are larger number than those of neighbors or time stamp reaches to the upper limit, finish this label propagation algorithm.
6. Otherwise, increment the time stamp ($t = t + 1$) and process vertices again.

4.2.5 Merge

Label Propagation function generates many small communities and they are restricted to be sub-graphs of *Ego Minus Ego* networks. In order to find the larger and global communities in the given whole graph, they will be merged in *Merge* function.

In *Merge* function, generated communities by *Label Propagation* will be merged. Parameter ϵ is a given threshold indicating overlaps between communities. They will be merged if and only if at most the ϵ of the smaller one is not included in the bigger one. When $\epsilon = 0$, two communities will be merged only if one of them is a proper subset of the other, and when $\epsilon = 1$, they will be merged together even these communities do not share any nodes. In general, two overlapped community will be more likely merged if the value of ϵ is large.

In this example described in Figure 4.8, two communities are overlapping each other. The left one has 4 nodes, the right one has 6 nodes and 2 vertices are overlapped. In this situation, they are 50% ($\frac{2}{4}$) overlapped. If $\epsilon = 0.6$, at least 40% vertices of the smaller community must

be overlapped to be merged, then these communities will be merged as a community with 8 vertices. On the other hand, if $\epsilon = 0.3$, at least 70% vertices must be overlapped for merge, then they will not be merged.

The *Merge* function also randomly chooses an existing community to be chosen next by shuffling orders of communities to prevent choosing the same communities in every iterations, which causes unequal output of communities.

4.2.6 Performance Characteristics of DEMON algorithm

Coscia et. al. [10] evaluated their original DEMON algorithm with real-world networks in Java implementation. In these performance evaluations, they used Amazon data set with 410,236 vertices and 2,439,437 edges. They mentioned the core of DEMON algorithm (*Ego Minus Ego* and *Label Propagation*) took less than a minute, while Merge function with increasing thresholds elapsed one minute to one hour.

In Congress and IMDb data sets, the original DEMON program detected the better community in F-Measure than other referred algorithms (HLC [59], Infomap [58], Modularity [60] and Walktrap [61]). However, they did not evaluate more detailed DEMON's performance comparing with those algorithms or network data.

4.3 IncrementalDEMON - Proposed Algorithm

We propose an incremental version of DEMON algorithm named "Incremental DEMON" that incrementally detects communities over time from streaming data. Incremental DEMON is comprised of three functions: incremental *Ego Minus Ego* network extraction and modification, incremental *Label Propagation* and optimized *Merge* function.

Figure 4.1 and 4.2 are the pseudo codes of the original DEMON algorithm and IncrementalDEMON algorithm.

4.3.1 Incremental Functions Description

In the original DEMON algorithm, intermediate networks and communities are generated for all vertices (ego) in the given graph from the first state. However, most of the results of *Ego Minus Ego* extractions, *Label Propagation* and *Merge* procedures from the incrementally updated network are the same as from based networks if there are few differences between

Require: Input data graph $G : (V, E)$, community set $C = \phi$, overlapping threshold $\epsilon \in [0, 1]$
Ensure: Overlapping community set C

```
for v ∈ V do
    e ← EgoMinusEgo(v, G)
    C(v) ← LabelPropagation(e)
    for c ∈ C(v) do
        c ← c ∪ v
        C ← Merge(C, c, ϵ)
    end for
end for
return C
```

Figure 4.1: Original DEMON algorithm retrieved from [10]

Require: Input data graph $G : (V, E)$, community set $C = \phi$, overlapping threshold $\epsilon \in [0, 1]$, added/removed vertex set in this step ΔV
Ensure: Overlapping community set C

```
for v ∈ ΔV do
    e ← IncrementalEgoMinusEgo(v, G)
    C(v) ← IncrementalLabelPropagation(e)
    for c ∈ C(v) do
        c ← c ∪ v
        C ← IncrementalMerge(C, c, ϵ)
    end for
end for
return C
```

Figure 4.2: IncrementalDEMON algorithm

network difference like only additional a couple of vertices or edges with the real social network growth in a few seconds.

In this situation, it is time-wasting to repeat same graph processing every time and impossible to catch up a real-time network growths. To overcome this problem, we proposed an incremental version of DEMON by introducing incremental algorithms for these three functions, and enabled it to minimum modification for existing communities. We will show incremental versions of them.

4.3.2 Incremental Ego Minus Ego Function

In the incremental situation, An added vertex or edge will affect only the neighbor vertices, and almost all *Ego Minus Ego* networks will not be changed before and after the incremental process. When a new vertex is added, we need only to re-construct *Ego Minus Ego* networks for the neighboring vertices of the newly added vertex like Figure 4.3. When a vertex is removed, we need only to remove a vertex and edges from *Ego Minus Ego* networks for the neighboring vertices of the removed vertex.

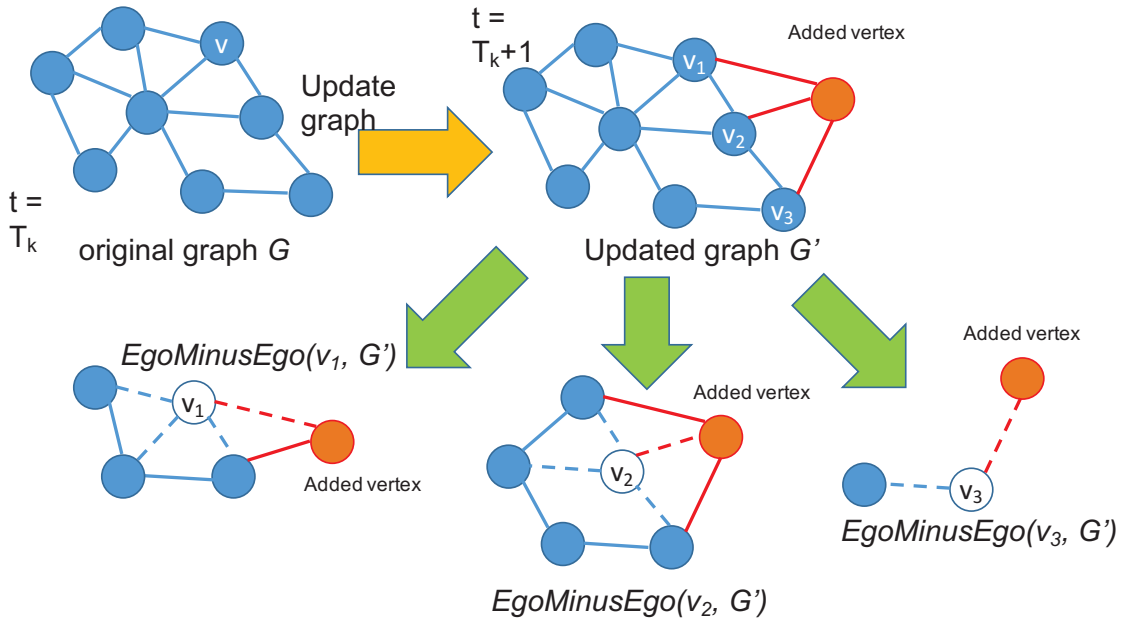


Figure 4.3: An Example of Incremental *Ego Minus Ego* (add single vertex and neighboring edges).

The pseudo code of the incremental *Ego Minus Ego* function is shown in Algorithm 4.5. When single vertex and connecting edges to neighbors are newly added, The algorithm is repeated for each added edge. Compared with the original *Ego Minus Ego* function constructing *Ego Minus Ego* networks for all vertices (Algorithm 4.4), the incremental function only updates two vertices for origin and destination of an newly added edge.

Require: $G : (V, E), v \in V$
Ensure: $G' : (V', E')$
  for all $v \in V$ **do**
    $V' \leftarrow neighbors(v)$
    for all $v' \in V'$ **do**
      for all $e' \in outer(v')$ **do**
        if $target(e') = v$ **then**
          $E' \leftarrow E' \cup e'$
        end if
      end for
    end for
  end for

Figure 4.4: Original *Ego Minus Ego* algorithm

4.3.3 Incremental Label Propagation Function

The incremental label propagation for each extracted *Ego Minus Ego* network will perform independently. At *Label Propagation* algorithm, a frequently appeared vertex is selected or if the frequency is the same, and then it randomly selects a vertex. Repeat the *Label Propagation* algorithm only near the added / removed vertex. Because the idea of label propagation method in original DEMON is based on [25], which chooses and sets the label with highest frequency in neighbor vertices, it is natural to set the highest frequently appeared labels to those added vertices.

An example of incremental *Label Propagation* procedure with addition a vertex is shown Figure 4.6.

Require: $G : (V, E), e \in E$,
$EgoMinusEgo : G' = (V', E') \subseteq G$
Ensure: Updated G'
for all $v \in [source(e), target(e)]$ **do**
$V' \leftarrow neighbors(v)$
for all $v' \in V'$ **do**
for all $e' \in outer(v')$ **do**
if $target(e') \in V'$ **then**
$E' \leftarrow E' \cup e'$
end if
end for
end for
end for

Figure 4.5: Incremental *Ego Minus Ego* algorithm (add single edge)

4.3.4 Merge Function

DEMON algorithm applies "local-first" approach rather than top-down view. Generated communities from previous *Label Propagation* procedure are too small and local. In order to detect more global communities, all overlapping communities should be merged together. Needless to say, the result of incremental *Label Propagation* is incrementally changed, so we need to apply a little modification to the original result of *Merge* process.

The authors of [10] mentioned, however, the Merge function does not hold the incrementality property, and an alternative simpler algorithm keeps incrementality were needed as their future work.

On the other hand, an alternative Merge function can be defined to combine the results provided by the core of the algorithm, thus preserving its possibility to scale up in a parallel framework. We cannot adjust the result of merge function only by adding or removing subgraphs because the decisions whether some communities should be merged would change and future scenarios might be different.

On the other hand, most communities with no common vertices do not have any relationships with added single vertex, and we can reduce the number of comparisons between an incrementally updated community and other communities. In order to specify communities which each vertex belongs to, we defined a mapping table between vertex ID and set

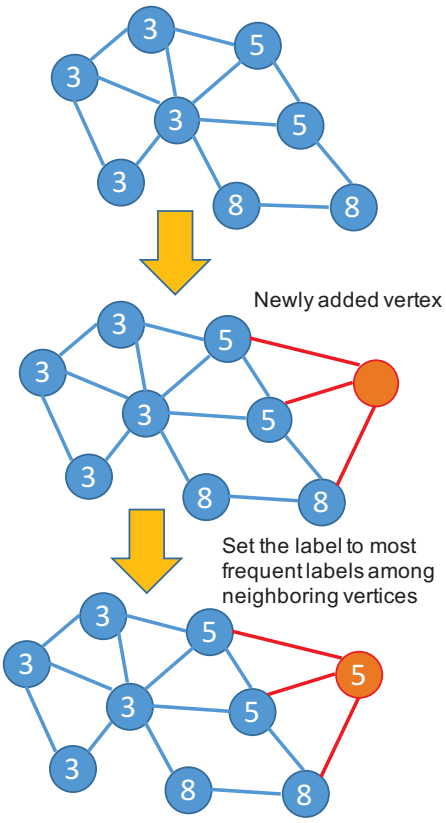


Figure 4.6: Incremental Label Propagation.

of community ID. Elements of the table will be constructed and refereed during the initial *Merge* function and the construction cost of this table object can be ignored in the incremental process by reusing it.

We describe optimized merge function to Algorithm 4.7.

Require: C : Total community set,
c : Updated community,
T : Mapping table,
ϵ : Threshold
Ensure: Updated C and T
$Cset \leftarrow \phi$
for all $v \in c$ **do**
$Cset \leftarrow Cset \cup T[v]$
end for
$merged \leftarrow False$
for all $c' \in Cset$ **do**
if $c'.size \leq c.size$ *and* $c' \subseteq_\epsilon c$ **then**
$u \leftarrow c' \cup c$
$C - c',\ C - c$
$C \leftarrow C \cup c$
$merged \leftarrow True$
Return
end if
end for
if $merged = False$ **then**
$C = C \cup c$
for all $v \in c$ **do**
$T[v] \leftarrow T[v] \cup c$
end for
end if

Figure 4.7: Pseudo Code of Optimized *Merge* Algorithm with Community Mapping Tables

In the first for-all loop, this function extracts possible communities to compare. Keys of the mapping table T are vertex IDs and values are sets of community ID which the vertex belongs to ($T[v]$ means the community ID sets with vertex v). At the end of this for-all loop, the community set $Cset$ stores the all possible community IDs which might be merged to the

given community c.

These communities are compared in the second for-all loop one after another. The inside of this loop is the exactly same as the comparison procedure of *Merge* function in original DEMON algorithm. If a community should be merged is found, merge these communities and finish this optimized *Merge* function. If there are no communities to be merged, register the given community c as the new global community and update this mapping table.

In fact, our optimized method has extra cost for constructions and references the mapping table. However, the number of comparison and merge process were reduced by referring this table. While original *Merge* function compared all-to-all small communities even most communities are never overlapped, our optimized *Merge* function compares only overlapping communities, then it eliminates unnecessary merging repetition.

Furthermore, we can reduce the community comparison processes in the *Merge* function of our IncrementalDEMON algorithm. In order to find communities we should actually compare quickly, we defined two types of tables. The first table represents mapping between vertex ID and community IDs the vertex belongs, and the second one represents the number of vertices and overlapping communities for each community. These tables will be updated when vertices or edges are incrementally added to or deleted from communities.

We will show an example of the way with how incremental *Merge* processing with updates of these tables when an edge is added (Figure 4.8(.

1. Suppose an edge between v_4 and v_6 is added, and v_4 is about to join community C_2 by the result of incremental *Label Propagation* function.
2. Update these tables describe member vertices and overlaps according to updated vertices and communities where they belongs.
3. Apply *Merge* function only to communities which of statuses are updated.
4. If they are actually merged, update these mapping tables again.

4.3.5 Computation Complexity

Suppose the degree of the updated (added or removed) vertex v is k_v and the average degree of graph is k. n and m is the number of total vertices and edges of a given graph. We also define $|C|$ as the number of generated communities.

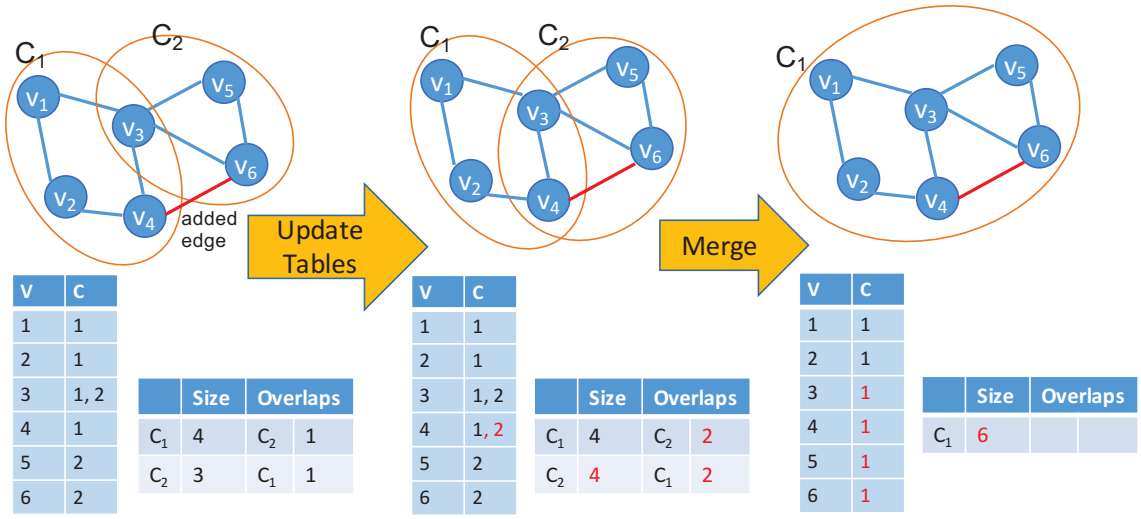


Figure 4.8: An Example of Incremental Merge Processing with Optimized Data Structures.

4.3.6 Ego Minus Ego Network Extraction

The *Ego Minus Ego* function extracts *Ego Minus Ego* networks from neighbor vertices of the update vertex, and then add or remove the neighbor vertices. From the discussion, the computation complexity for an *Ego Minus Ego* function is $O(k_v \times k)$.

Considering that *Ego Minus Ego* networks must be stored for all vertices, the space complexity is $O(n+m) \times k$. That means it needs large heap memory space to store the number of average degree of a graph, and it is impossible to manage billion-scale networks in this method. We will need a scalable method such as data optimization and leveraging distributed-memory environments for our future work to handle larger-scale data.

4.3.7 Label Propagation

The *Label Propagation* function puts or updates a label to the newly added vertex. The computation complexity is $O(k_v)$ because it only updates the added vertex label from neighbor vertices.

The space complexity is only $O(n \times k^2)$, because the function stores the member of communities for each *Ego Minus Ego* network. The required heap memory space might be much larger than *Ego Minus Ego* network extraction.

4.3.8 Merge Function

In the original DEMON, the computation and space complexity in the *Merge* function is $|C|$ and $|C| \times n$. On the other hand, our optimized *Merge* function, the computation and space complexity is up to $|k_v|$ and $|k_v| \times n$ because we have only to compare near communities from neighbor vertices.

4.4 Performance Evaluation

4.4.1 Implementation

We implemented original and incremental DEMON algorithms in C++ on top of our own graph database called IBM System G [62]. The original DEMON algorithm has been implemented as a Python script and can be downloaded from DEMON homepage [63], and we ported this script to C++ code.

4.4.2 Experimental Data Set

In order to compare our proposed incremental method and the original one, we obtained the same data sets as the experiment of the authors of [10] did as follows.

Congress
: The network of legislative collaborations between US representatives of the House and the Senate during the 111st US congress. It is a relatively dense network, with 536 vertices and 14,198 edges, and average degree is 53.98.

IMDb
: We also used the part of the data set of actors who star in at least two movies during the years from 2001 to 2010, filtering out television shows, video games, and other performances. The number of vertices and edges are 56,542 and 185,247. The average degree is 6.55.

Amazon
: As a larger real network, we used Amazon purchases data. It has frequent co-purchases of products are recorded for the day of May 5th 2003. The number of vertices and edges are 410,236 and 2,439,437, and the average degree is 11.89.

Input files for the experiments have edge list in the CSV format where each line is comprised of one edge from a source vertex to a destination vertex. In order to simulate time-evolving dynamic graph, we have 2 different files - one for baseline graph data and another file for dynamically added edge list. Base graph data to be added will be read at once and additional edge data to be added incrementally.

4.4.3 Experimental Environment

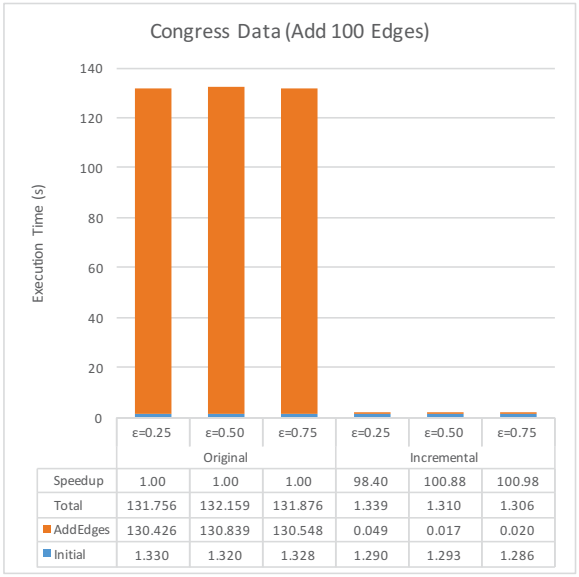
In the original DEMON algorithm, we executed the whole algorithm for dynamic networks. We generated each network input file by newly added edge data one by one, then start DEMON program and measure execution time and obtain communities. On the other hand, our incremental DEMON algorithm loads a baseline graph data and calculate the initial community only once. We evaluated performance by executing the original and incremental DEMON algorithm on IBM System G process. The execution environment is 64bit Red Hat Linux, with single core Intel(R) Xeon(R) CPU E5-2660 @ 2.20GHz and 378GB random access memory.

4.4.4 Performance Results

The execution times for community detections with additional 100 edges with Congress, IMDb and Amazon data sets are shown in Figure 4.9, Figure 4.10 and Figure 4.11 respectively. These figures describe the elapsed time for the first step (the lower part of each bar) to generate and analyze baseline graph objects and the changed graph with additional new 100 edges to update and analyze them (the upper part of each bar).

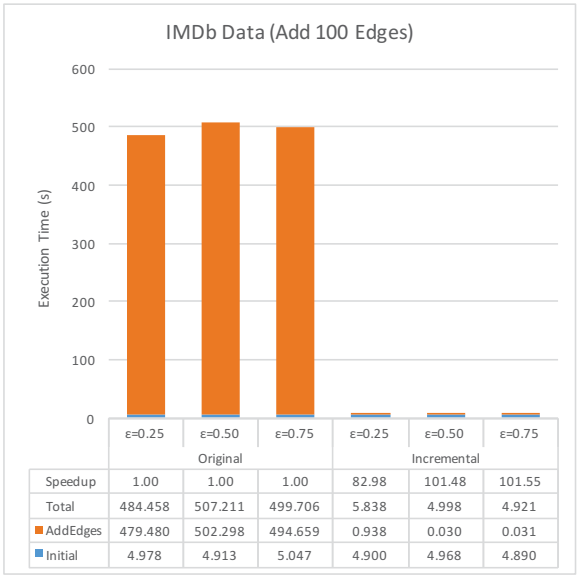
With Congress data set, the original DEMON algorithm took about 1.3 seconds in the first step and constantly more than one seconds in each step. The total execution time was about 130.4 seconds with the parameter ϵ used in *Merge* function was 0.25. In our incremental DEMON algorithm, it took about 1.29 seconds in the first step, almost same as the original algorithm. However, it took only 0.049 seconds for incremental steps to add 100 edges, resulted 1.34 seconds in total and 98.4 times faster than that of the original method.

In IMDb data set, the original algorithm took more time, about 4.79 seconds per each step, and the total execution time was more than 6 minutes. In our incremental DEMON algorithm, the first step took 4.868 seconds, but the incremental phase took only 0.96 seconds in the first step. The total execution time was only 5.828 seconds which achieved about 83.0 times faster with the parameter ϵ was 0.25. When ϵ was 0.5 or 0.75, our incremental method



	ε=0.25	ε=0.50	ε=0.75	ε=0.25	ε=0.50	ε=0.75
	Original			Incremental		
Speedup	1.00	1.00	1.00	98.40	100.88	100.98
Total	131.756	132.159	131.876	1.339	1.310	1.306
AddEdges	130.426	130.839	130.548	0.049	0.017	0.020
Initial	1.330	1.320	1.328	1.290	1.293	1.286

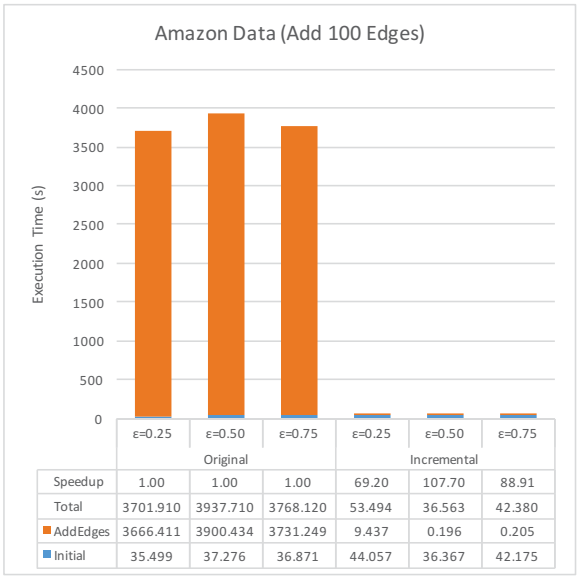
Figure 4.9: Elapsed time using Congress graph data when new 100 edges are added



	ε=0.25	ε=0.50	ε=0.75	ε=0.25	ε=0.50	ε=0.75
	Original			Incremental		
Speedup	1.00	1.00	1.00	82.98	101.48	101.55
Total	484.458	507.211	499.706	5.838	4.998	4.921
AddEdges	479.480	502.298	494.659	0.938	0.030	0.031
Initial	4.978	4.913	5.047	4.900	4.968	4.890

Figure 4.10: Elapsed time using IMDb graph data with additional 100 edges

was around 101.5 times faster than that of the original method.



	ε=0.25	ε=0.50	ε=0.75	ε=0.25	ε=0.50	ε=0.75
	Original			Incremental		
Speedup	1.00	1.00	1.00	69.20	107.70	88.91
Total	3701.910	3937.710	3768.120	53.494	36.563	42.380
AddEdges	3666.411	3900.434	3731.249	9.437	0.196	0.205
Initial	35.499	37.276	36.871	44.057	36.367	42.175

Figure 4.11: Elapsed time using Amazon graph data with additional 100 edges.

In Amazon data set, the effect of our incremental method was still remarkable. The original DEMON algorithm almost constantly took 36.6 seconds for each step, and total execution time was more than one hour. On the other hand, our incremental DEMON algorithm took about 44.1 seconds for the first step, but the incremental steps for additional 100 edges was only about 10.1 seconds. The total time was 54.2 seconds, 69.2 times faster than the original method.

The main reason why the incremental method took more time in the first step with Amazon data set than with Congress and IMDb data is that the number of vertices and edges are much larger, and the overheads for inner rearranging intermediate data structures such as C++ standard map structures became visible.

Even if the threshold parameter ϵ for the *Merge* function was changed, the total time for DEMON algorithm was not drastically changed. Generally, if ϵ is large, small communities would be frequently merged and the communities for comparison would drastically decrease. On the other hand, when ϵ is small, the number of merge processes will decrease, but many communities overlapped but not merged must be compared for every iterations. These comparison procedures could be reduced if we introduce additional data structure for comparison tables like in Figure 4.8. It will be our future work.

4.5 Related Work of Incremental Community Detection

Zhenyu and Ming propose a scalable community detection method for tag assignments stream clustering (TASC)[64]. The method requires not only network structures, but needs tag set and user data for all network vertices. They used three types of data sets with about 10,000 to 70,000 vertices and 50,000 to 440,000 tag assignments, and no initial edges were specified. Their performance experiment showed that it took about 10 ms to 30 ms with one assignment.

Sheng et. al. focused an incremental label propagation algorithm and added the time sequences as well as labels to vertices [65]. The average computation complexity was the same as the total number of local edges connected to the local vertices $O(m)$ (m: the number of edges), and the worst case of the overall time was $O(m+n)$ (n: the number of vertices). The authors showed that the average iteration time was 4.411 seconds with on-line game players networks. In this algorithm, additional tag data were necessary and the final results of network division might change with the number of iterations. Our proposed method considers only network structures, and output results are stable.

Jingyong and Hongsheng proposed an incremental algorithm based on growing social networks considering that most communities tended to evolve gradually over time[66]. They used computer-generated data sets and internet peer-to-peer network data with 10,876 vertices and 39,994 edges for their experiments. Compared with RI (based on Radicci method) and IC (finding nodes that are connected with increments: almost same as this method), the computation time was almost same as the IC method, and better than the other methods especially increasing the total number of vertices.

4.6 Summary

We proposed an incremental community detection algorithm based DEMON algorithm for real-world network analyses. We also showed our algorithm reduced the execution time of community detection to less than single second for incrementally updated networks, and makes around 100 times faster than original method in 100 steps iterations.

Besides the speed-up of community detections for incrementally growing networks, it took much more time to generate *EgoMinusEgo* networks and set labels for all these networks in the first step. Moreover, it consumed much larger heap memory space than these original three functions we mentioned and it was not realistic to apply it for much larger networks with a standalone commodity machine with limited memory size.

As a future work, we will optimize *EgoMinusEgo* network extractions and storing functions, and improve this method on parallel and distributed-memory machines with running incremental community detections. We will only show the result on a single node, but as our next work, we will implement a distributed version of our proposed method.

In this research, we implemented and evaluated incremental method only for adding edges. We will also propose additional incremental procedures for deleting edges and vertices as another future work.

We will also need more precise evaluations about outputs like distributions of generated community sizes and precision. Original DEMON mentioned that the distribution was better than that of other popular community detection methods. Because our method especially *LabelPropagation* is supposed approximation to set labels, the final result may be different from the original results.

Chapter 5

Incremental Graph Pattern Matching

Graph pattern matching algorithms to handle million-scale dynamic graphs are widely used in many applications such as social network analytics and suspicious transaction detections from financial networks. On the other hand, the computation complexity of many graph pattern matching algorithms is expensive, and it is not affordable to extract patterns from million-scale graphs. Moreover, most real-world networks are time-evolving, updating their structures continuously, which makes it harder to update and output newly matched patterns in real time. Many incremental graph pattern matching algorithms which reduce the number of updates have been proposed to handle such dynamic graphs. However, it is still challenging to recompute vertices in the incremental graph pattern matching algorithms in a single process, and that prevents the real-time analysis. We propose an incremental graph pattern matching algorithm to deal with time-evolving graph data and also propose an adaptive optimization system based on reinforcement learning to recompute vertices in the incremental process more efficiently. Then we discuss the qualitative efficiency of our system with several types of data graphs and pattern graphs. We evaluate the performance using million-scale attributed and time-evolving social graphs. Our incremental algorithm is up to 10.1 times faster than an existing graph pattern matching and 1.95 times faster with the adaptive systems in a computation node than naive incremental processing.

5.1 Introduction

5.1.1 Background

Graph pattern matching is a fundamental graph data processing method. It extracts subgraphs with some restrictions such as topologies or vertex and edge attributes represented as a small graph **(query pattern graph)** from a large-scale graph **(input data graph)**. Graph pattern matching algorithms are widely used in many applications, such as social network analysis [37] and suspicious financial transaction detection [67]. In social network analysis, for example, graph pattern matching algorithms enable the extraction of particular organizations and clubs of members and relationships with typical roles. In suspicious financial transaction detection, it allows frequent screenings of bank accounts involving suspicious transactions. These networks typically used in these real-world applications are usually million-scale, and their structures update frequently.

5.1.2 Problem Definition

There are several significant challenges to apply graph analytics to time-evolving graphs.

First of all, real-world graphs update their topologies and attributes of their vertices and edges frequently. Many graph analytics needs to follow the update, but it is hard to re-compute for the entire graph after every change to avoid missing newly matched patterns.

It is also difficult to apply a pattern-matching algorithm to full graph structures with million edges because it has at least polynomial time complexity. In particular, the computation complexity of subgraph isomorphism is NP-complete [68]. In order to run graph pattern matching in realistic execution time, we need to adjust the computation area and adopt an approximate algorithm with less computation complexity.

5.1.3 Requirements of Graph Pattern Matching

Optimization methods for graph pattern matching algorithms have been proposed to deal with these problems. However, some of them are not suitable for real-world network analysis because they do not support topological matching and vertex or edge attribute matching. There are pattern matching algorithms which support exact topological matching, but it takes a long time to update results incrementally even it outputs few matched patterns with the time constraint.

To propose a graph pattern matching method addressing these problems, we define the following requirements of our proposed graph pattern matching method. These requirements are fundamental to conduct flexible and real-time graph pattern matching for real-world applications.

Subgraph isomorphism: There are graph pattern matching algorithms such as graph simulations which do not consider the topological matching in order to reduce the computational cost of subgraph isomorphism. However, for many applications, such topological restrictions restrict the patterns that are worth exploring.

Attributes of vertices and edges: Vertices and edges in real-world graph data usually have attributes such as labels and properties. In financial transaction data, for example, vertices (bank accounts) have account ID, customer name, or account type while edges (remittances) have transaction date and amount. Users extract patterns with restrictions of such attributes.

To address this issue, we add to our graph pattern matching method a requirement to support the label and property filtering for both vertices and edges. In other words, when the vertices and edges in a query graph have labels and properties, we require that the corresponding vertices and edges in the matched subgraphs also have labels and properties.

Approximate patterns should also be found: As we mentioned at the first requirement, topological restrictions are necessary for many graph pattern matching applications. On the other hand, some applications favor outputs of topologically approximate patterns. For example, in a suspicious transaction pattern detection from financial transaction graphs, it is unrealistic to prepare all criminal transaction patterns and match them exactly. There might be potentially suspicious transaction patterns similar to obviously suspicious patterns. In this situation, our proposal allows to topologically approximate patterns as well as exact patterns.

5.1.4 Research Challenges and Contributions

Many matching problem for large-scale and time-evolving graphs have been proposed. Previous work proposed in [37, 69, 70, 71] updates the results partially by the update of input data graph structures and attributes. However, the efficiency of the incremental process depends on the types of the input data graphs and query pattern graphs.

We define the following research challenges.

- How can we reduce computation costs of an approximate subgraph isomorphism algorithm for dynamic graphs? Can we update the partial result of graph pattern matching efficiently?
- Can we adjust the incremental processing from the type of input data graphs and query pattern graphs automatically?
- Can we make the optimization method applicable to various input graphs and query patterns?

In this work, we propose an adaptive and incremental graph pattern matching algorithm with reinforcement learning.

First, we propose an incremental graph pattern matching **(IGPM)** algorithm based on an existing approximate graph pattern matching algorithm named G-Ray [6]. G-Ray supports a "best-effort" topological pattern matching with reduction of the computation complexity. It has core functions to search for the best matched vertices and edges. We extend the incremental version of these functions according to the type of graph updates: edge additions, edge removals and vertex label updates.

We also propose "partial execution manager" **(PEM)** to choose subgraphs for re-computation by our incremental graph pattern matching algorithm. The best matching vertices set depends on several factors including the number of vertices and edges in the input data graph and required time limit. It partitions input data graphs into small clusters and output vertices in the updated clusters. The PEM component has a reinforcement learning sub-component to receive feedback such as elapsed time and the number of matched patterns. This information is used to adjust these parameters of the graph clustering.

In summary, our technical contributions are as follows.

1. We propose an incremental graph pattern matching **(IGPM)** algorithm based on an approximate and fast graph pattern matching algorithm to reduce re-computation cost.
2. We propose a performance optimization component named partial execution manager **(PEM)** for **IGPM** to output the best vertex set for re-computations in the incremental processing.
3. We show the efficiency of our algorithm and system using some input data graph and query pattern graph types commonly used in real-world graph data sets and pattern matching applications.

4. We evaluate the performance of **IGPM** and **PEM** using real-world time-evolving graph data sets and several types of query pattern graphs in a multi-core computation node.

5.2 Related Work

Previous works proposed graph pattern matching algorithms which deal with large-scale graphs. Some of them relax topological restrictions of subgraph isomorphism due to the high computation complexity, and others apply clustering algorithms to input data graphs to prune unnecessary re-computation areas.

5.2.1 Graph Pattern Matching for Large-scale Graphs

Graph pattern matching algorithms are categorized to *subgraph isomorphism* [72] and *graph simulation* [34]. The main difference is whether it requires topological matching including edges or it considers only corresponding vertices. Subgraph isomorphism extracts subgraphs with the same graph topology as the query graph. On the other hand, graph simulation relaxes the topological restriction and only take vertex matching into considerations.

Because most real-world applications require the topological constraints, subgraph isomorphism is a better fit as it can extract specific patterns by specifying topological constraints. However, because the computational complexity of subgraph isomorphism problem is NP-complete [68], many relaxed pattern matching algorithms have been proposed [37] [39] [73].

The computation cost is linear with the number of matched patterns; hence it is unrealistic to extract all possible patterns. [74] [75] [76] [77] extract only top-k patterns. These methods do not guarantee to extract all possible patterns, but they consider the topological matching to a certain degree.

Some graph pattern matching algorithms for large-scale and attributed graph are proposed with the graph data optimization techniques such as query graph decomposition [75] [78] [79] and input data graph compression [80] to reduce the computation costs keeping the output qualities.

5.2.2 Incremental Graph Pattern Matching

Table 5.1 represents the state-of-art incremental graph pattern matching algorithms recently proposed for large-scale graph data. It also represents the compatibility with the requirements

we defined in Section 5.1.3. Columns "ISO" (subgraph isomorphism), "Attr" (vertex and edge attributes), "Time" (real-time processing) and "Approx" (approximate matching) indicate whether these methods fulfill each requirement. Our system with the incremental graph pattern matching algorithm and an optimization component for incremental process supports all four features.

Paper	Method	ISO	Attr	Time	Approx
IncGM+	Compute fringe subgraphs	✓	✓		✓
D-ISI	Distributed graph pruning	✓	✓		✓
Bounded Simulation	Graph compression			✓	✓
$Turbo_{ISO}$	NEC & COMB/PERM	✓	✓	✓	
(Our Method)	IGPM-PEM	✓	✓	✓	✓

Table 5.1: The state-of-art incremental graph pattern matching methods

Abdelhamid et al. proposed IncGM+ [69], a heuristic subgraph isomorphism algorithm for large evolving graphs. It separates the input data graph into frequent and infrequent updated subgraphs and prunes the update area by adjusting the boundary subgraphs named "fringe". They also proposed the optimization of subgraph embeddings in the fringe to reduce the memory overhead for storing fringe subgraphs. It reduced the computation time keeping the small memory overhead, but some data sets did not improve the performances because of too much graph pruning were performed.

Wickramaarachchi et al. proposed a distributed graph pruning algorithm for dynamic graphs (*D-IDS*) and a distributed incremental subgraph isomorphism algorithm (*D-ISI*) [70]. *D-IDS* limits the search space for subgraph isomorphism with small diameter graphs. This method can extract exact patterns with lower latency in small-diameter input graphs, but the latency is much longer in the large-diameter graphs.

Fan [37] proposed a notion of graph pattern matching the revised incremental version of "Bounded Simulation" [35] for social network data with optimizations by query preserving graph compression and distributed graph pattern matching. However, even though it considers the connectivity of the two matched vertices connected each other, the limitation is still permissive, and it does not evaluate the similarity between the query pattern and extracted patterns.

Han et al. proposed an optimized incremental subgraph isomorphism algorithm named $Turbo_{ISO}$ [71]. It uses region exploration by the neighborhood equivalence class (NEC) and

combine and permute (COMB/PERM) techniques to prune unnecessary searches.

5.3 Proposed Method

To process incremental graph pattern matching efficiently, we propose an adaptive graph pattern matching algorithm named IGPM-PEM, with incremental graph pattern matching (IGPM) algorithm and partial execution manager (PEM) to re-compute the updated subgraphs.

In this section, we present our proposed IGPM algorithm and PEM for re-computation optimization. First, we introduce an incremental and approximate subgraph isomorphism algorithm based on G-Ray [6] algorithm. Next, we introduce PEM to optimize the IGPM consists of reinforcement learning and graph clustering components. Finally, we discuss the efficiency of several types of input data graph and query pattern graph qualitatively.

Figure 5.1 represents our system with IGPM-PEM component. It iterates our incremental graph pattern matching algorithm, reinforcement learning, and graph clustering processes to optimize the performance of graph pattern matching.

First, when some topological or attribute updates occur in the input data graph such as edge additions or vertex label changes in the IGPM component, the PEM component inputs the accumulated features of the data graph such as the number of vertices and edges. It also inputs the elapsed time of the IGPM phase and the number of extracted patterns and their similarities as rewards of the reinforcement learning.

The reinforcement learning component computes the best granularity of subgraphs for the re-computation process from the reward and input parameters such as input graph data size. It sends the minimum number of the subgraph size to the graph clustering component.

The graph clustering component partitions the input data graph into the clusters with the appropriate size. It repeats the Louvain method [19] until these partitioned clusters cannot be divided further or the size of clusters is smaller than the parameter defined at the reinforcement learning component. If a cluster has at least one updated graph element such as an added edge, all vertices in the cluster will be subject to the re-computation of IGPM in the next iteration. Finally, PEM sends the resultant vertex set to the IGPM component.

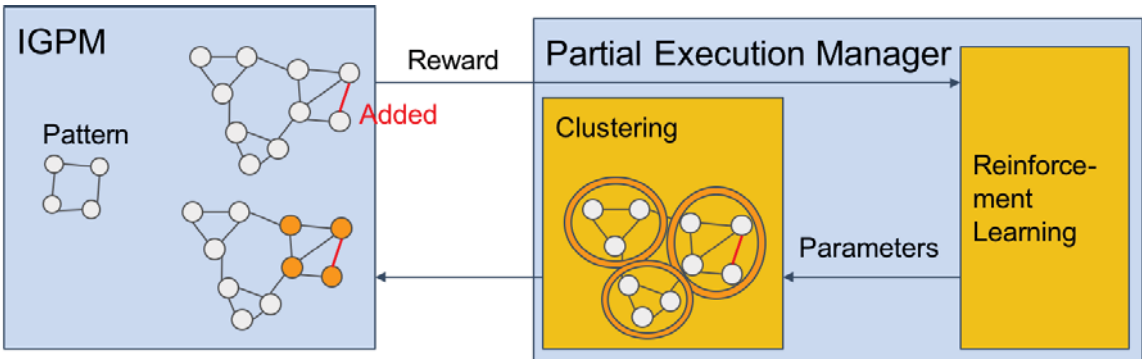


Figure 5.1: Components of our adaptive IGPM system

5.3.1 Approximate Graph Pattern Matching

Previously we mentioned the requirements of graph pattern matching that topologically matches exact patterns and similar patterns should be extracted from the input data graph. For this reason, we extended an existing approximate subgraph isomorphism algorithm which extracts both exact and approximate patterns.

G-Ray [6] is an approximate subgraph isomorphism algorithm which extracts subgraphs matched with the desirable patterns which are "best-effort results". That means each vertex in the subgraph corresponds to a vertex in the query graph, and the connectivity is maintained even when the exact pattern does not exist in the input data graph. Additionally, the computation cost of the G-Ray is linear concerning the size of the data graph compared to the polynomial complexity of the exact subgraph isomorphism algorithms.

We adopt G-Ray as a baseline of our method since it has following properties, which satisfy part of the requirements we defined.

Approximate subgraph isomorphism: It exacts the most proper subgraphs for the query pattern as subgraph isomorphism algorithms. When it finds matched edges directly connecting two matched vertices, it will choose the most suitable one with the highest goodness score.

Vertex attribute matching: It finds each matched vertex from input data graph with the same vertex label as in the query pattern graph.

Extract a specified number of best-effort subgraphs: It extracts similar patterns as well as exact patterns by the best-effort matching process. Users can specify the number of patterns to extract.

Linear computation complexity: Graph pattern matching algorithms based on subgraph isomorphism are NP-complete, which is prohibitive for large-scale graphs. On the other hand, the computation complexity of G-Ray algorithm is the linear of the number of vertices in the data graph.

G-Ray has the following three core functions to find the best matched first vertex named "seed" vertex, neighbor vertices of the previously matched vertices, and edges connecting these matched vertices.

Seed-finder: It finds the most suitable vertex (seed vertex) from the input data graph, producing the largest goodness function score corresponding to a query graph with the same label.

Neighbor-expander: It finds the best matching neighbor vertex from the input data graph of the previously matched vertex, producing the best goodness function from the input graph to indicate the closeness of two vertices.

Bridge: It finds the best path (one or more connecting edges) between these two matched vertices with the derivative of the EXTRACT [81] algorithm, and connects them from the input data graph with intermediate edges.

The two matched vertices are not always neighboring each other because there are no such directly connecting edges between these vertices which exist in the query graph. Even in this case, G-Ray finds the best (usually the shortest) paths in the bridge function and output the "best-effort" approximate query.

5.3.2 Incremental Graph Pattern Matching

To execute IGPM efficiently of time-evolving graphs, we design the incremental version of the G-Ray algorithm by extending these three core functions of the original G-Ray. Figure 5.2 shows the algorithm of IGPM.

The incremental extension of seed-finder function performs different procedures by the change of graph structures or attributes. It re-computes the RWR scores and goodness scores of vertex pairs should be directly connected. Then, it iterates the incremental extension of "neighbor-expander" and "bridge" functions for each vertex as a "seed" connected to the added or removed edges.

The incremental extension of neighbor-expander function updates the best-matched neighbor vertex of the previously matched vertex. When the added edge is connected to the "seed"

Require: The input data graph G, the query pattern graph H_q, vertices list V_l from updated edges

Ensure: Updated subgraphs H_l

for $v \in V_l$ **do**
  Update all RWR values $r_{v,u}$ from v to reachable vertices u
end for
for $v \in V_l$ **do**
  $seed \leftarrow v$
  repeat
    for $u \in neighbors(v)$ **do**
      Compute goodness function g_u from $r_{v,u}$
    end for
    $u' = argmax_u(g_u)$
    Update the best path by **bridge** algorithm with $r_{v,u}$
    Mark the edge (v, u) in H_q as "processed"
  until All nodes in H_q are "processed"
  Add the extracted pattern to H_l
end for

Figure 5.2: IGPM Algorithm

vertex, it re-computes the goodness function of the seed vertex and all the neighboring vertices. If some different neighbor vertices produce the higher goodness function score than that score of the current neighbor vertex, replace it with the new vertex with the highest score.

When the seed vertex or the neighbor vertices are updated after applying the incremental seed-finder or incremental neighbor-expander, it needs to re-compute the original bridge function with these updated matching vertices. If these origin and destination vertices the same as in the previous state, it only checks the updates of intermediate vertices and edges, and re-computes and reconnects the path partially. When an edge is added to the input data graph, it extracts all existing paths passing the source and destination vertices and re-computes goodness score and these intermediate paths between them if needed.

5.3.3 Partial Execution Manager

To execute our IGPM algorithm with real-world dynamic graphs more effectively, we also propose **Partial Execution Manager (PEM)**, an adaptive vertex assignment component for dynamic graphs. PEM computes a vertex set for re-computation from the current graph states and parameters.

It consists of two sub-components: the graph clustering component and the reinforcement learning component. The reinforcement learning component adjusts the parameter of the graph clustering component which determines the area of subgraphs to extract vertices for re-computations through the Louvain method.

Figure 5.3 shows the optimized IGPM algorithm with PEM. Lines from 7 to 20 represents the PEM process: reinforcement learning and graph clustering are performed at line 7 and 13.

Re-computation Vertices Extraction

In incremental graph pattern matching, we need to re-compute neighboring vertices of the updated area to find new patterns because even an edge update will affect the result of RWR in the neighboring vertices. On the other hand, it is inefficient to re-compute all unaffected vertices to find patterns with only a few vertex or edge updates. To extract re-computation vertices, determining the subgraph areas is important. PEM adjusts the area adaptively using graph clustering and reinforcement learning to extract new patterns.

When too many vertices updated to be re-computed all vertices, PEM extracts the mini-

Require: The input data graph G, the query pattern graph H_q, initial community size c, total number of steps T

Ensure: Subgraph patterns H_l

1: $step \leftarrow 0$
2: **repeat**
3: $V_l \leftarrow$ vertices from updated edges at $step$
4: $H_l \leftarrow IGPM(G, H_q, V_l)$
5: $t \leftarrow$ elapsed time of $IGPM$
6: $x \leftarrow$ number of vertices and edges of G
7: $y \leftarrow$ reinforcement learning with input x and reward $\frac{1}{t}$
8: **if** $y == 0$ **then**
9: $c \leftarrow c - 1$
10: **else**
11: $c \leftarrow c + 1$
12: **end if**
13: $communities \leftarrow$ clustering with recursive Louvain from G into communities with size c
14: $l \leftarrow$ vertices from updated edges at $step + 1$
15: $V_l = list()$
16: **for** $com \in communities$ **do**
17: **if** com and l has common vertices **then**
18: Add all vertices in com to V_l
19: **end if**
20: **end for**
21: **until** $step > T$

Figure 5.3: Optimized IGPM with Partial Execution Manager

mum number of vertices from fine-grained subgraphs. The reinforcement learning component outputs the smaller number of community size for the graph clustering component to output the fine-grained vertex set.

Then, the graph clustering component repeats the Louvain methods recursively until each size of clusters is less than the specified size threshold and outputs small communities. Finally, PEM extracts vertices from these communities, and output all vertices in each community if at least one of vertices in the community also belongs to the updated vertex set.

Data Graph Clustering

PEM has a graph clustering component to divide input data graph into small clusters and extract vertices for re-computation from the several clusters where edge updates occur.

We use the Louvain method [19] as a clustering algorithm. It is a heuristic community extraction method with optimizing modularity. It iterates changes of communities locally to maximize modularity and aggregates these communities.

The main reason we use the Louvain method as clustering in PEM is that we can adjust the size of communities by the number of iterations. When it has insufficient time and does not want so many patterns, it partitions the graph into fine-grained clusters and output minimum vertices for re-computations. Moreover, it can handle larger (million-scale) graphs in the reasonable computation cost and the additional time for the graph clustering process itself is relatively small.

Reinforcement Learning

There are many factors to find the best subgraph area for re-computations, such as the input data graph and query pattern graph, updated edges and so on. However, it is difficult to find out the best parameter because there is no answer data (the best parameter) for unknown data sets (input data graph and query pattern graph), and it is also hard to learn parameters for all combinations of data sets.

To avoid the problem, we used the reinforcement learning model to adjust parameters of graph clustering component automatically. The main reason we adopt reinforcement learning is that the reward and action patterns are clear to be defined, while the primary graph features to affect the performance of IGPM are not obvious.

PEM has reinforcement learning component to adjust the parameter. It updates its learn-

ing model to output the best parameter during the iteration of IGPM and PEM process, so we do not have to try many parameters and formulas of the model manually.

We use a Deep-Q Network (DQN) [82] for our reinforce learning model. The observation of the reinforcement learning agent, i.e., input layer of this DQN, are two-dimensional real-valued variables: the density of input data graph (a proportion of the number of edges out of the number of vertices) and the proportion of the affected communities. The DQN has two hidden layers with four units, and each layer is fully connected. The output layer to determine an action has two units. The action is to increment or to decrease the threshold of minimum community size.

5.3.4 Qualitative Discussion of Graph Types

In graph pattern matching applications with real-world data set, users may want to try several kinds of queries according to the kind of applications or data sets. In this section, we discuss the combinations of input data graph and query pattern graph types in detail to prove the validity of our method using PEM qualitatively.

Input Data Graphs

Considering our method is used in primary large-scale real-world applications, first we discuss the validity for the following five types of input data graph and kind of clustering algorithms.

1. Scale-free graph
2. Random graph
3. Sparse graph (with isolated edges)
4. Sparse graph (with dense subgraphs)
5. Dense graph

Scale-free graph: Scale-free graphs are considered as the representation of real-world networks. They have a few hub vertices and many leaf vertices and usually appear in many real-world domains such as social networks. However, clustering algorithms which maximize the modularity are not suitable for such graphs. Since the degree distribution is unbalanced,

clustering algorithms usually output unbalanced clusters, and it is difficult to adjust these sizes. The cluster size depends on the degree of hub vertices.

Random graph: Since the degree distributions of random graphs are relatively decentralized, compared to scale-free graphs, the result of clustering can be more balanced and flexible.

Sparse graph with isolated edges: Sparse graphs appear as the initial states of real-world networks such as social network and bank transactions. These type of graphs usually have many isolated vertices, edges and tiny components, which should be excluded in advance to apply clustering operations to large components.

Sparse graph with dense subgraphs: When a sparse graph is growing to a certain degree, it usually has dense subgraphs. Such a graph is the most suitable graph type for clustering. It is also suitable for dense, cycle query graphs, but not for line query graphs. Line patterns found in previous steps will be disconnected by clustering.

Dense graph: Since there are many edges compared to the number of vertices, it has high computation costs for re-computation of G-Ray, particularly about neighbor-expander and bridge functions. Also, because many edges are disconnected by a clustering algorithm such as Louvain method, IGPM may miss many patterns. A minimum-cut clustering algorithm such as METIS [26] may provide better results for such graph type. Such clustering algorithms will output unbalanced clusters and results to less efficiency, but it minimizes the number of new missed patterns.

Query Pattern Graphs

Not only input data graphs but query pattern graphs have many structure types, and the compatibility depends on the structure of input data graphs. To explain our adaptive IGPM system is effective for various query pattern graphs as well as input data graphs, we need to confirm the efficiency of significant types of query patterns one by one.

1. Star query
2. Cycle query
3. Dense query

Star query graphs consist of a single high-degree ”hub” vertex and many ”leaf” vertices which directly connect to the hub vertex. Such hub vertices exist in the scale-free, random and

dense graphs or clusters, so it can be applied these input data graphs. In our IGPM algorithm, the "seed" vertex of the G-Ray is always "hub" vertex because high-degree vertices derived from RWR usually have high goodness score in the G-Ray algorithm. When we use star query graphs in our IGPM derived from the G-Ray algorithm, we have only to find and re-compute subgraphs with hub vertices with the similar degree to the hub vertices in query graphs. That means it has only to re-compute hub vertices in many cases because other leaf vertices can never be "seed" vertices.

Cycle query graphs are also sequences of vertices and edges such as line query graphs, but the structure is closed. Such patterns are used for some specific applications. For example, they can detect transaction loops involving money laundering in bank transaction graphs and suspicious corporation relationships in finance risk detection applications. The diameter of the cycle query is relatively large so that when graph clustering process divides dense input data graphs and many edges are disconnected, some pattern subgraphs crossing clusters will be missed. In dense graphs, IGPM of cycle query graph is difficult because it needs to update long matched cycles. However, there are some hints to choose vertices for re-computation. First, all vertices must be at least 2-degree. Second, hub vertices are likely to be part of cycles. By the hints, we can set the priorities to high-degree vertices for re-computation.

Dense query graph means a query graph with many edges compared to the number of vertices because our IGPM algorithm needs to repeat neighbor-expander and bridge functions many times. One or more vertices in a dense graph have high degree and usually connected to all other vertices. There are some features of the pattern matching using dense query graphs. First, it can match only with dense graphs (or dense subgraphs). Moreover, such as cycle query graphs, when many edges are disconnected by a clustering, some patterns will be missed.

Summary of the qualitative discussions

Based on these above discussions, the compatibility of input data graphs, graph clustering algorithms and query pattern graphs is the Table 5.2. "Min-Cut" or "Max-Mod" in the "Clustering" column means that minimum cut (e.g. METIS [26]) or modularity maximization (e.g. Louvain [19]) is preferable.

Input Data Graph	Clustering	Star	Cycle	Dense
Scale-free	Min-Cut	✓	✓	
Random	Max-Mod	✓	✓	
Sparse, isolated edges	Max-Mod		✓	
Sparse, dense clusters	Max-Mod	✓	✓	✓
Dense	Min-Cut	✓		✓

Table 5.2: Compatibility of input data and query pattern graph types and clustering methods

5.4 Evaluation

To verify the effect of our IGPM-PEM system, we conducted performance evaluations with real-world graph data sets. We also validate the qualitative discussion in section 5.3.4.

5.4.1 Implementation

We implemented batch and incremental graph pattern matching algorithm, Partial Execution Manager in Python 2.7.13 and the NetworkX 2.1 package for graph processing. As the reinforcement learning framework in PEM, we used keras-rl [83], TensorFlow 1.7.0 and Keras 2.1.5 and these Python interfaces. We also used NumPy package (version 1.14.2) for RWR computation with 28 hardware threads. Figure 5.4 represents the software stack of IGPM and PEM.

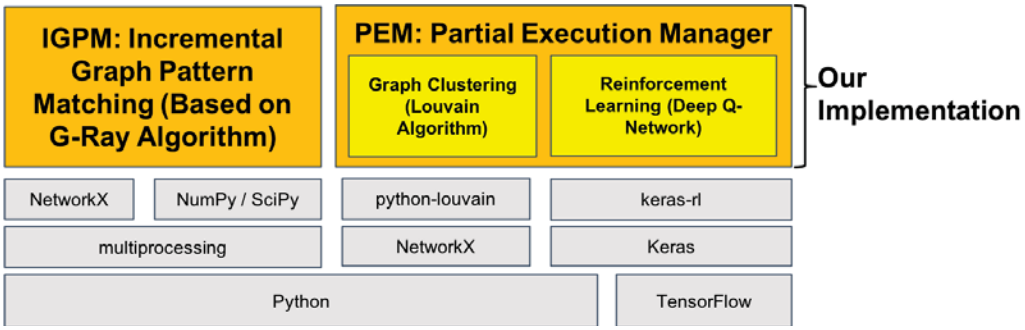


Figure 5.4: Software stack of IGPM and PEM. We implemented IGPM and PEM components in Python.

5.4.2 Computing Environment

We conducted all performance evaluations on one of parallel and distributed compute nodes in TSUBAME 3.0 supercomputer in Tokyo Institute of Technology. Each compute node has Intel Xeon E5-2680 V4 Processor CPUs (Broadwell-EP, 2.4GHz, 14 cores, 2 sockets), 4 Tesla P100 for NVLink-Optimized Server GPUs and 256GB (DDR4-2400 32GBx8) RAM.

5.4.3 data sets and Scenario

We used temporal graph data from Harvard Dataverse [84] and SNAP Network data sets [85]. Table 5.3 is the list of temporal graph data we used in the experiments. The column "Steps" means the number of graph structure updates when edges are added. In the graph data "friends2008", we extracted all friendship edges created in 2008 from friends network [84], and set the step value from the hour of each timestamp. In other graph data, we extracted all edges with timestamps and set the step value from the day.

Name	Category	Vertices	Edges	Steps
friends2008 [84]	Social	224,879	3,871,909	6,893
transactions [84]	Social	112,130	538,597	1,779
sx-askubuntu [85]	Webpage	159,316	964,437	2,060
sx-mathoverflow [85]	Webpage	24,818	506,550	2,350

Table 5.3: Temporal Graph Data Sets Used in Our Experiments

As query pattern graphs, we used the following type of small graphs: triangle, square, star and complete graph.

1. Triangle (loop graphs with 3 vertices)
2. Square (loop graphs with 4 vertices)
3. Star query graph with 5 vertices (4 leaf vertices)
4. Complete graph with 4 vertices and 6 edges

In the reinforcement learning component of PEM, we used *Epsilon-greedy* policy as the decision making policy from unit values of the output layer of the DQN. This policy selects an action corresponding to the unit with the highest state-action value(greedy action) with

probability $1 - \epsilon$, and it chooses an action uniformly at random with probability ϵ. In the following experiments using reinforcement learning, we set the value of ϵ to 0.5.

We measured elapsed time and the number of re-computed vertices for 10 steps, from 101 steps to 110 steps. We started the measurements of them after edge additions for 100 steps because the initial state of input data graphs do not have enough edges and too sparse to extract many patterns.

In the following experiments, we compared the three graph pattern matching algorithms: batch (re-compute graph pattern matching process from scratch for each step), naive incremental (incremental pattern matching using results of the previous step, with the fixed community size) and adaptive incremental (incremental pattern matching with adaptive community size).

5.4.4 Performance of the naive IGPM

Figure 5.5 shows the elapsed time of the batch (Batch) and naive incremental (Inc) graph pattern matching with the square query pattern graph. The highest speedup ratio against the batch processing is 9.98 in friends2008 data set. On the other hand, the ratio in sx-mathoverflow and sx-askubuntu is 3.10 and 3.37. The efficiency of our naive incremental algorithm depends on the data graphs, but at least 3 times faster than the batch version in these data sets.

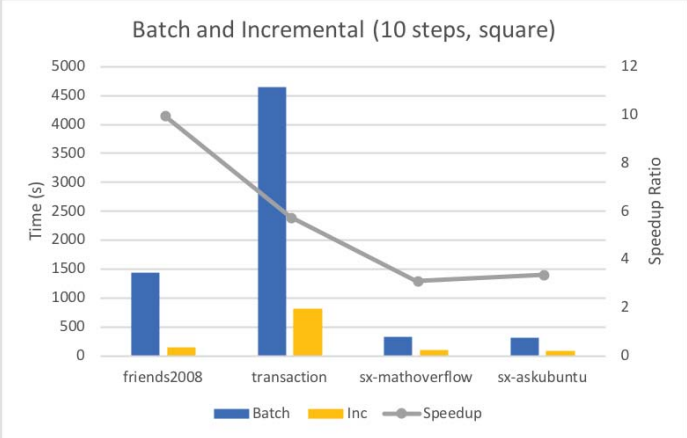


Figure 5.5: Elapsed time and speedup ratio of the batch and the naive incremental graph pattern matching with a square query

Figure 5.6 shows the total elapsed time and speedup ratio of the batch (Batch) and naive incremental (Inc) graph pattern matching using friends2008 data set and four query pattern graphs. Overall, the naive incremental version is from 9.5 to 10.1 times faster than the batch version. The speedup ratio is stable in the same data graph.

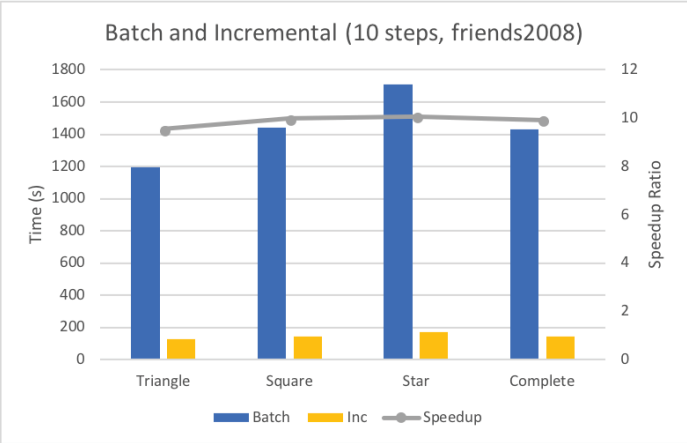


Figure 5.6: Elapsed time and speedup ratio of batch and naive incremental graph pattern matching with friends2008 graph

5.4.5 Performance of the adaptive IGPM with PEM

Figure 5.7 shows the total elapsed time and speedup ratio of naive and adaptive incremental graph pattern matching with the square query pattern graph. The adaptive version is from 1.17 to 1.96 times faster than the naive version.

Figure 5.8a shows the total elapsed time and speedup ratio of naive and adaptive incremental graph pattern matching using friends2008 data set and four query pattern graphs. The speedup ratio of adaptive version is at most 1.17. Figure 5.8c shows the total elapsed time and speedup ratio of naive and adaptive incremental graph pattern matching using sx-mathoverflow data set and four query pattern graphs. The speedup ratio of adaptive version is around 1.9 for all query patterns.

From the result of Figure 5.8, the efficiency of our adaptive incremental pattern matching algorithm also depends on the type of data graph rather than query patterns.

In friends2008 data set, the total number of re-computed vertices is 37,018 in the batch version, which is 14.8 and 13.0 times as large as naive and adaptive incremental version.

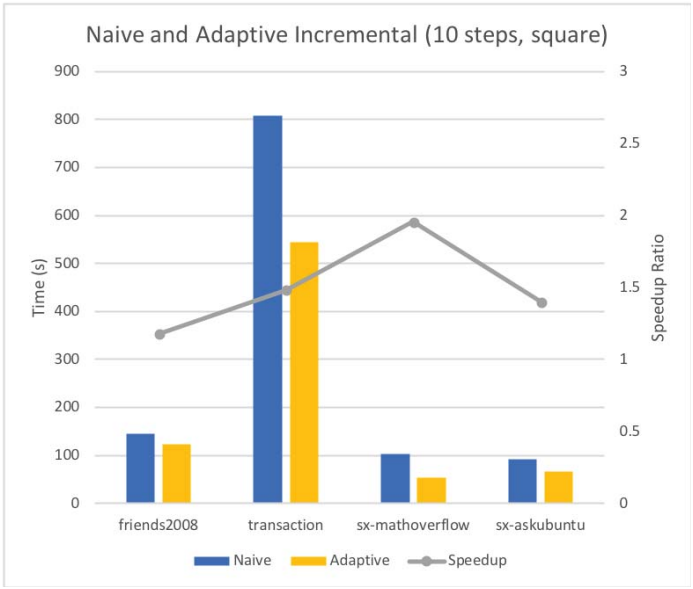
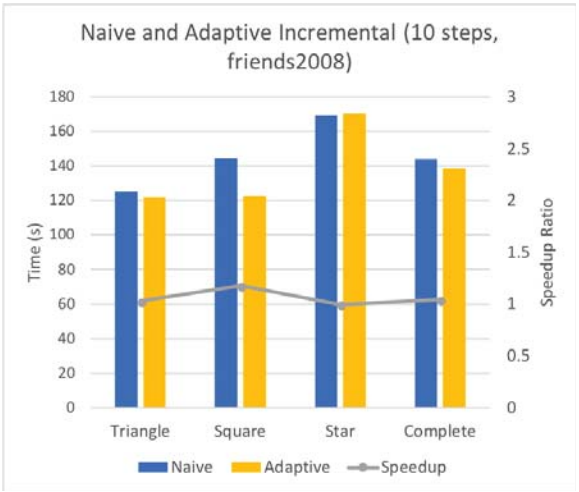


Figure 5.7: Elapsed time and speedup ratio of naive and adaptive incremental graph pattern matching with a square query

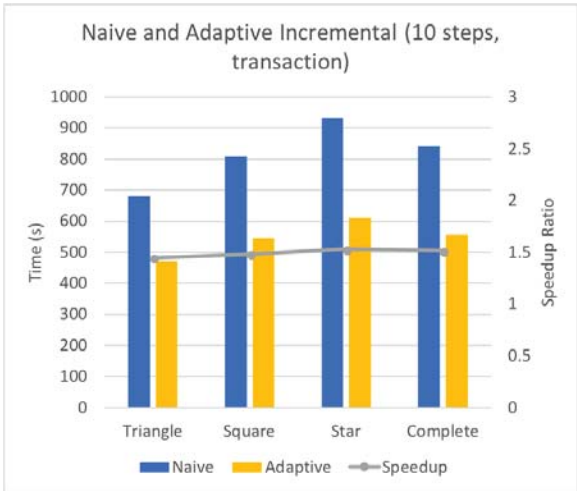
Because the computation time of our graph pattern matching algorithm depends on the number of vertices for re-computation, the ideal speedup ratio should be 14.8 in the incremental processing. In transactions data set, on the other hand, the total number of re-computed vertices in the batch version is 6.97 and 9.00 times as large as that of incremental version. The speedup ratios of naive and adaptive are 5.80 and 8.08 respectively, which is better than that of friends2008 data set. There are several reasons the actual speedup ratio is smaller than the ideal value in the data set. First, the RWR computation cost is still significant for dense graphs. When some vertices in large dense subgraphs are re-computed, it needs to update RWR results of all vertices in these subgraphs, which reduces the benefits of incremental computations.

5.4.6 Precision Evaluation

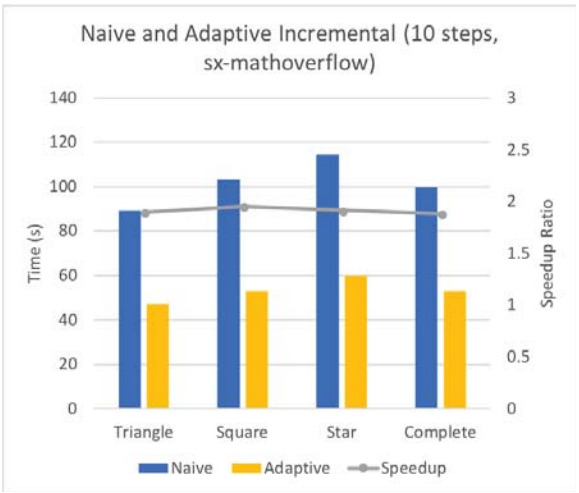
Figure 5.9 shows the total number of patterns correctly extracted from the friends2008 graph. Except for the star query graph, the naive and adaptive incremental version extracts from 37% to 73% additional patterns. Figure 5.10 shows the total number of patterns correctly extracted with a square query graph. The adaptive incremental version extracts 25% to 73% additional patterns. The reason more patterns can be extracted in the IGPM is that our



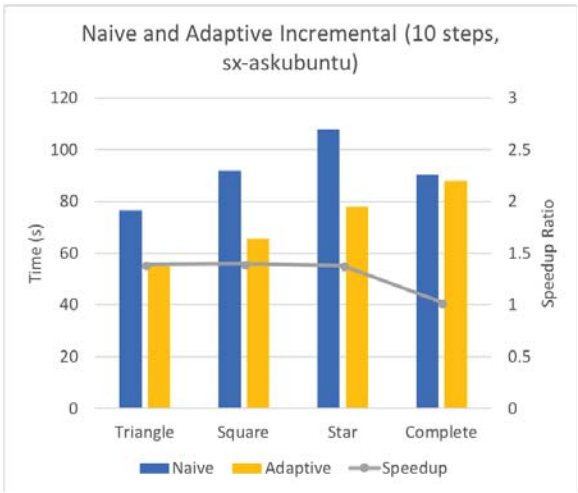
(a) friends2008



(b) transactions



(c) sx-mathoverflow



(d) sx-askubuntu

Figure 5.8: Elapsed time and speedup ratio of naive and adaptive incremental graph pattern matching

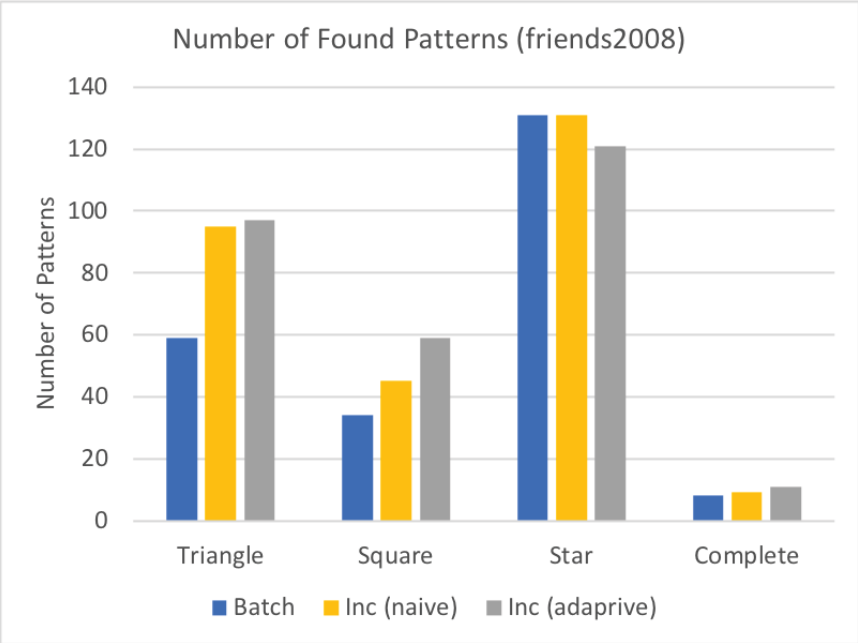


Figure 5.9: Number of found patterns from friends2008 graph

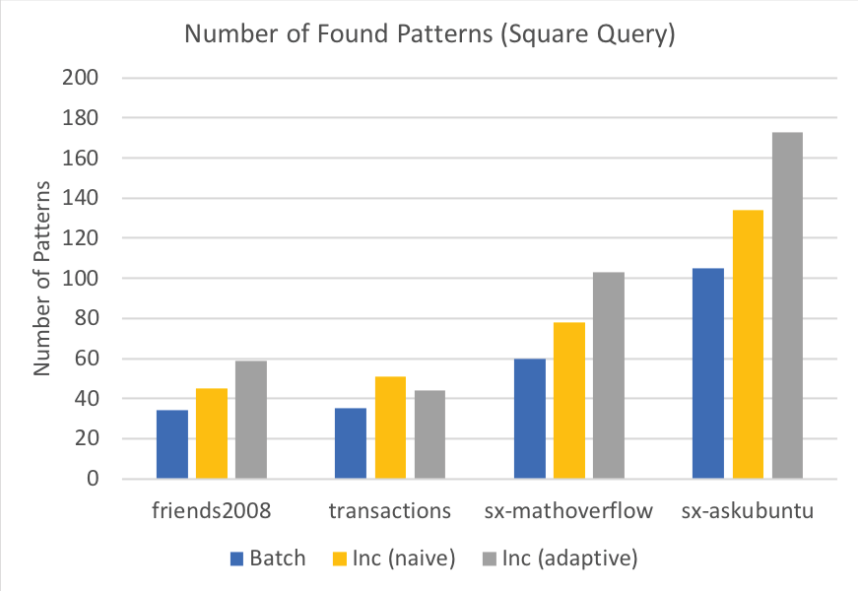


Figure 5.10: Number of found patterns with a square query

incremental algorithms re-compute all updated vertices as "seed" vertices and repeat the G-Ray algorithm respectively, and therefore these algorithms extracted more patterns which the batch version could not find.

5.5 Discussion

5.5.1 Reinforcement Learning for Adaptive Incremental Pattern Matching

In adaptive graph processing frameworks, one of the approaches is pre-executions or profiling before actual executions. From the sampled profiling or log data, it estimates the best parameters and update strategies.

The adaptive agent assignment method in the agent-based traffic simulation we proposed in Chapter 3 is the similar approach. Our method requires pre-executions to obtain the distributions of running vehicles in each snapshots, and then it defines the schedules of adaptive vehicle migrations.

Schiller et, al. proposed frameworks [86, 87] to estimate the most efficient data structure from the size, structure, update frequency of the given static and dynamic graphs. The estimation is performed while compiling and profiling, and then it re-compiles the source code to fit the hardware. This framework will work on the same data sets and hardwares, but it will need to the profiling and re-compilation again for new data sets.

In this solution, the cost of pre-execution or profiling must be small enough compared to the actual executions. Moreover, even it work well for the same data sets, scenarios and computational environments, some different situations will not work well.

In order to apply the optimized algorithms and parameters to various data sets and hardwares, we should use machine-learning approaches. In machine learning executions, Although the machine learning approach also requires pre-executions for learning phase and it is hard to generate more accurate parameters compared to the profiling approach, it can adopt to new data sets and environments.

We used a reinforcement learning approach to extract matching patterns with the best-effort throughput (largest number of patterns found within the unit of execution time). The advantage of reinforcement learning compared to supervised learning is that we only have to define the reward (the goal of incremental graph pattern matching) and we do not need to know the best parameter in advance.

In our proposed PEM model, the input parameters are only the number of vertices and the number of edges in the entire graph, and the throughput as the reward.

In order to send better feedback from the result and environment of IGPM to the PEM framework, we used the scale of the data graph (number of vertices and edges) as the input and the throughput defined above as the reward of reinforcement learning.

5.5.2 Graph Clustering Algorithms

We adopted the Louvain method as a graph clustering algorithm to extract vertices for re-computations in the IGPM process so that the PEM can adjust the size of these vertices set accurately. We discuss the requirements of graph clustering algorithms for adaptive incremental graph analytics and some representative graph clustering algorithms which satisfy these requirements.

The requirements of graph clustering for adaptive incremental graph analytics are following:

Adjustable Cluster Size by Parameters The main purpose of graph clustering is to minimize the number of vertices for re-computation with keeping the higher accuracy. By adjusting the size of graph clustering outputs, we can extract the only minimum number of vertices.

Small Computation Complexity Compared to the IGPM processing, the computation complexity of PEM itself must be small. The computation complexity of exact clustering algorithms usually too high to apply every iterations, so we need an approximate clustering algorithm.

Following these requirements, the Louvain method is actually suitable for the graph clustering in PEM. Then, we list some representative graph clustering algorithms (label propagation, METIS and spectral clustering) and discuss what kind of graph clustering algorithms are suitable for adaptive graph pattern matching.

Label propagation [25] has small computational complexity (linear to the number of vertices and edges), and the granularity can be adjustable by the number of iterations.

METIS [26] is also fast graph partitioning algorithm with a coarsening heuristics technique.

Spectral clustering [27] consists of two phases; computing eigenvectors of the graph Laplacian and k-means clustering. Because the output eigenvectors of the first phase is determin-

istic, we need to specify the number of clusters in the k-means clustering phase.

5.6 Summary

We proposed an adaptive IGPM system with an approximate IGPM algorithm and PEM for adaptive assignment of vertices for re-computations. We also discussed the qualitative effect of our system with various input data graph type, query pattern type, and graph clustering algorithms. Then, we designed and implemented the whole system and components. Finally evaluated the efficiency of our IGPM algorithm and PEM component, and then showed up to 10.1 times speed-up in IGPM against the original G-Ray algorithm, and 10.8 times faster in our adaptive and incremental system with PEM.

We still have room for improvements of the IGPM and PEM. First, we discussed the compatibility of input data graph type and query pattern type, but we did not conduct all experiments for all combinations of these graph types. We excluded line query graphs due to the poor assumption of qualitative discussion in section 5.3.4, but we will run experiments for such inefficient patterns in the future.

We could improve the reinforcement learning model of the partial execution manager. We used only the number of vertices and edges from an input data graph as input parameters of the DQN, but it will extract many patterns more efficiently if we use detailed graph structure or other features of graphs. That might increase the computation cost for reinforcement learning due to the size of DQN, but it can improve the overall efficiency of the IGPM.

In this work, we implemented the proposed system in Python. However, it causes performance overheads and cannot handle large data sets and number of update steps than we used. As future work, we plan to implement it in C++ to make maximum use of more efficient parallel and distributed libraries such as OpenMP and MPI and computing environments such as multi-core CPUs and GPUs to handle larger graph data in parallel.

Chapter 6

Overall Discussion

We proposed several performance optimization methods for specific agent-based traffic simulation, community detection algorithm, and graph pattern matching algorithm. However, we have a question remaining about the applicability of these methods for other simulation frameworks, graph algorithms and data sets. Even we considered it for each proposed method to the specific traffic simulation application or a graph algorithm individually, we did not still discuss whether our adaptive and incremental methods can be applied to other agent-based simulation applications such as a money transaction simulation and other graph algorithms such as the shortest pathfinder.

6.1 Applicable Graph Algorithms

In this section, we discuss the applicabilities, limitations and other requirements for generalizations of our proposed methods.

First, we make a quantitative discussion about the applicabilities of our adaptive object re-assignment and parameter adjustment methods in traffic simulation. Based on these procedures of our proposed methods, we discuss additional requirements and generalizations for other agent-based simulation frameworks and applications.

Next, we discuss the possibility of generalizations and applications about our proposed incremental community detection algorithm. Some community detection algorithms have different requirements from the baseline algorithm we used. We verify the applicability of several other community detection algorithms considering the properties or requirements each

algorithm has (e.g., whether it allows overlapping communities or not, and what kind of metrics for detected communities such as modularity and other requirements and metrics it optimizes).

Finally, we discuss the requirements and limitations of our proposed adaptive and incremental graph pattern matching with reinforcement learning for generalization. As many adaptive graph analytics algorithms have been proposed for dynamic graphs, we evaluate the qualitative effectiveness of our method in these adaptive algorithms.

6.1.1 Adaptive Object Reassignments in Traffic Simulations

We discuss the applicability of our adaptive method to other agent-based frameworks and traffic simulation applications.

There are many adaptive load-balancing methods for agent-based simulations, and most of them assume the continuous spaces geographically partitioned into squares. For example, Cosenza et, al presented a load balancing schema for distributed agent-based simulations [88]. It assumes particle simulations and partitions the space in a certain dimension, and then assigns them to MPI processes.

On the other hand, our optimization application assumes traffic simulations, and the running vehicles move along the road network. Unlike in particle simulations, it needs to consider the road network structure and the trip of each vehicle to balance the workloads.

We used Megaffic [7], which is also an agent-based traffic simulation application for our performance optimization methods such as adaptive agent object re-assignments. However, Megaffic is different from other agent-based traffic simulations in the following aspects.

First, the road network as a simulation environment forms graph structure rather than continuous space used for particle simulations. In this case, an appropriate graph clustering algorithm is necessary to partition road networks for better load balancing.

Second, the unit of agents are not only vehicles but also cross points. Vehicle and cross point agents exchange messages every simulation steps about its statuses such as its position and speed. Subsequently, vehicles which receive these messages update its position and speed. In this case, when some vehicles migrate to other partitions, the workload for all partitions will be gradually unbalanced. To reduce the load unbalancing, dynamic re-partitioning of the road network and re-assignments of vehicle agent can be approaches to solve it.

6.1.2 Adaptive Adjustment of Synchronization Intervals

In distributed simulations, synchronizations and communication among computation processes are fundamental to keep the consistency of whole simulations. There are two types to manage them; frequently synchronizations for each simulation step, and asynchronous processing.

In the simulation framework we implemented named XAXIS, it uses a centralized simulation manager for global synchronizations. One of the simulation processes manage simulation steps, and some processes need to wait until all other processes finish this simulation step. In this approach, the synchronization overhead such as waiting time will be significant when the workload of each process becomes gradually unbalanced. On the other hand, because the message exchanges including vehicle migrations are performed only once per simulation step, the communications among simulation processes are simple and relatively lower cost.

Hanai et, al. proposed an adaptive resource provisioning method and an asynchronous migration mechanism [89] to reduce the effects of load imbalance in Megaffic. When some vehicles are ready for migrations, it sends the vehicle objects and messages between vehicles and cross points asynchronously. They also proposed a staged asynchronous migration method to reduce its overhead in [90].

6.1.3 Incremental Community Detection

We proposed an incremental community detection algorithm (IncrementalDEMON) based on an existing algorithm by extending the three core functions of the based algorithm with incremental processing features: *EgoMinusEgo*, *LabelPropagation* and *Merge*. We discuss the applicability of the algorithm extension approach for other graph algorithms.

First, the incremental *EgoMinusEgo* function updates subgraphs for all neighbor vertices from the updated (added or removed) vertices. When a hub vertex is removed, the many neighbor vertices will be affected to construct the EgoMinusEgo networks.

The incremental *LabelPropagation* also updates the labels of neighbor vertices from the updated vertices. As we discussed in the section 6.5, the output of this function may differ to the original *LabelPropagation* when only part of the data graph is updated.

The incremental *Merge* function is more complex compared to the *EgoMinusEgo* and *LabelPropagation* functions. When at least one of small subgraphs are updated in *EgoMinusEgo* or *LabelPropagation*, the incremental *Merge* function need to compare and merge these can-

didate subgraphs again.

6.1.4 Incremental Graph Pattern Matching

Besides incremental graph pattern matching algorithm, we also proposed an incremental algorithm about graph pattern matching (IGPM-PEM). This approach of the IGPM is similar to the IncrementalDEMON in that we extended several core function of the base algorithm to support incremental processing. G-Ray, which is the base graph pattern matching algorithm of IGPM, consists of three core functions: *SeedFinder*, *NeighborExpander* and *Bridge*.

In order to make this discussion more simple, we generalize our IGPM method. Based on the G-Ray [6] we adopted as the baseline algorithm of IGPM, the graph pattern matching is processed the following direction.

1. Find the first matched vertex with the same label (Seed-Finder)
2. Find the second matched vertex neighboring the previously matched vertex (Neighbor-Expander)
3. Connect the matched vertex pairs with edges (Bridge)

With this notion of the graph pattern matching process, we explain how our proposed method can be applicable to graph pattern matching algorithms.

In subgraph isomorphism algorithms, each vertex in the query pattern graph and the input data graph must be matched one-by-one, and if a vertex pair in the query pattern graph are connected with an edge, the corresponding vertex pair in the input data graph must be also connected with an edge.

In graph simulation algorithms, each vertex pair connected by an edge must be matched, and vertices in the matched subgraph must be matched only about the vertex label.

Some graph pattern matching algorithms use query-decomposition approach, which extracts matched subgraphs of the decomposed query pattern graphs and then merge them into matched patterns. STwig [75], SJ-Tree [78] and MC-GPM [91] adopt this approach to reduce the computation complexity for large-scale graphs.

6.1.5 Partial Execution Manager for IGPM

Besides IGPM method, our proposed IGPM-PEM has the PEM component consists of graph clustering and reinforcement learning to adjust the number of re-computation vertices adaptively. In this section, we discuss what kind of input parameters and rewards are necessary to adopt PEM for various adaptive graph pattern matching algorithms.

In our proposed PEM, the input parameters are only the number of vertices and edges, and the reward is defined as the throughput (the number of patterns found per unit of time).

6.2 Limitations of Our Methods by Input Data Sets and Parameters

In the previous section, we discussed the applicabilities of our proposed method in the aspect of the properties of other graph algorithms. On the other hand, effectiveness and applicabilities of our adaptive and incremental optimization methods might be depend on the input data such as the scale of the road network in traffic simulations and query pattern graphs for graph pattern matching. In this section, we make quantitative discussions about the effectiveness and limitations of our proposed methods by the input data including parameters and scenarios.

6.3 Performance Modeling

In this section, we discuss the total execution costs with performance modeling. Then, we evaluate the effectiveness of our adaptive and incremental methods from the formulated result of the performance model.

6.3.1 Agent Migrations and Worker Reassignments in the Traffic Simulation

Because our implementations of Megaffic and XAXIS are based on Java, we should consider the overhead of runtime Java object managements such as serialization, deserialization and garbage collections.

In [86] Benjamin et, al. proposed framework for dynamic graph analysis in Java from data size, structure, read and update frequency in the runtime.

To discuss the qualitative effectiveness, we formularize the total computation time as follows. Table 6.1 shows the symbols for the formularization.

Symbol	Description
T	Total number of simulation steps
N_p	Number of X10 processes (computation nodes)
N_t	Total number of X10 threads (for all X10 processes)
N_V	Total number of vehicles
N_{CP}	Total number of cross points
N_R	Total number of roads
t_{CP}	Execution time to process a cross point per step
t_V	Execution time to process a vehicle object (to send messages and adjust position and speed)
t_{mig}	Execution time to migrate a vehicle object to another X10 process (including serialize and deserialize of the Java object)

Table 6.1: Symbols for formularization of Megaffic/XAXIS performance

In the single process executions without any external communications and synchronizations, the minimum computation time T_{min} per step with the ideal load balancing is following.

$$T_{min} = t_{CP}N_{CP}/N_T + t_V N_V/N_T \tag{6.1}$$

In actual, it is very hard to keep the load balancing because the number of vehicle agents each thread handles changes dynamically by continuous movement of vehicles. Suppose some vehicles are running part of the road network handled by the thread A, and N_{mig} of them are about to move to another part of the network handled by the thread B. In this case, the execution time for the thread A (T_A) and B (T_B) will be the following formulas.

$$T_A = t_{CP}N_{CP}/N_T + t_V(N_V/N_T - N_{mig}) \tag{6.2}$$
$$T_B = t_{CP}N_{CP}/N_T + t_V(N_V/N_T + N_{mig}) \tag{6.3}$$

In this case, the simulation time per step will be longer than the ideal time by $t_V N_{mig}$.

In the multiple processes (nodes) execution, vehicle migration costs among processes must be taken into consideration. We show an example of the performance model in two-process

execution with vehicle migrations between them. Note that the the performance model in multiple processes is essentially same as that of two-process.

Assume the process A and B are ideally balanced with the same number of running vehicles. Before vehicle migrations, the computation time is same as 6.1. When T_{mig} vehicles migrate from the process A to B, the execution time in the next step for the process A (T_A) and B (T_B) will be the following formulas.

$$T_A = t_{CP}N_{CP}/N_T + t_V(N_V/N_T - N_{mig}) + t_{mig}N_{mig} \quad (6.4)$$
$$T_B = t_{CP}N_{CP}/N_T + t_V(N_V/N_T + N_{mig}) + t_{mig}N_{mig} \quad (6.5)$$

Note that $t_{mig}N_{mig}$ is the cost of migration itself. The simulation time in this step will be longer than the ideal computation time by $t_V N_{mig} + t_{mig}N_{mig}$.

Next, assume the simulation perform agent reassignments of cross points and vehicles after this migration. We suppose the result of reassignments is ideal, which balances the number of vehicles per process again. In this case, N_{mig} vehicles and N_{CPmig} cross points will be reassigned from the process B to A. The cost of agent reassignments is almost same as the vehicle migrations ($t_{mig}N_{mig}$) because the execution cost of cross point reassignments is much smaller than vehicle migrations. The execution time (includes the reassignment time) in the next step after this reassignment for the process A (T_A) and B (T_B) will be the following formulas.

$$T_A = t_{CP}(N_{CP}/N_T + N_{CPmig}) + t_V N_V/N_T + t_{mig}N_{mig} \quad (6.6)$$
$$T_B = t_{CP}(N_{CP}/N_T - N_{CPmig}) + t_V N_V/N_T + t_{mig}N_{mig} \quad (6.7)$$

The total overhead of the agent reassignments compared to the ideal balanced situation is $t_{CP}N_{CPmig} + t_{mig}N_{mig}$. After all, the agent reassignments will be effective if the cost of vehicle migration is larger than that of cross point reassignments ($t_V N_{mig} > t_{CP}N_{CPmig}$). In our implementation of Megaffic/XAXIS, the reassignment cost of a vehicle is much larger than that of a cross point ($t_V >> t_{CP}$), so the agent assignment contributes to the overall performance improvement.

6.3.2 Adaptive Parameter Adjustments

The total cost of agent-based traffic simulations mainly consists of the computational costs in each computational process, communication costs between processes for vehicle migrations and synchronization costs due to load imbalances among all processes.

First, the computational cost of a partition is depend on the number of agents (cross points and vehicles). Because the Megaffic application we used assigns worker threads to cross points and these cross points handle running vehicles, we need to predict how many vehicles are running in each cross point.

6.3.3 Incremental Community Detection

Suppose the number of vertices and edges in the data graph are defined as the table 6.2.

Symbol	Description
N_V	Number of vertices in the data graph
N_E	Number of edges in the data graph
k	Average degree (N_E/N_V)
k_v	Degree of the vertex v
K	Maximum degree
N_C	Number of communities at just after LabelPropagation
$\Delta v(t)$	Number of vertices added at the step t

Table 6.2: Symbols for formularization of the DEMON and IncrementalDEMON performance

First, we formulate computation costs of the core functions (*EgoMinusEgo*, *LabelPropagation* and *Merge*) of the original **DEMON** algorithm.

The computation costs of *EgoMinusEgo* and *LabelPropagation* functions are $O(k(N_V + N_E))$ and $O(N_V + N_E)$ respectively. In the *Merge* function, the computation cost is $O(N_C^2)$ because it has to compare all small communities generated at the *LabelPropagation*.

Next, we formulate the costs of our proposed method **IncrementalDEMON**. The computation costs of the incremental *EgoMinusEgo* and incremental *LabelPropagation* functions are $O(k\Delta v(t))$ and $O(k\Delta v(t))$ respectively because the number of updated EgoMinusEgo networks is at most $k\Delta v(t)$. The cost of the (optimized) incremental *Merge* function is $O(k^2\Delta v(t))$ because the optimized function has to compare only neighboring subgraphs around the updated $\Delta v(t)$ vertices.

6.3.4 Incremental Graph Pattern Matching

Suppose the number of vertices and edges in the data graph and query pattern graph are defined as the table 6.3.

Symbol	Description
N_V	Number of vertices in the data graph
N_E	Number of edges in the data graph
k	Average degree (N_E/N_V)
N_{QV}	Number of vertices in the query pattern graph
N_{QE}	Number of edges in the query pattern graph
$n_V(t)$	Number of vertices for re-computations at the step t

Table 6.3: Symbols for formularization of the G-Ray and IGPM performance

Here are formularized computation costs of the G-Ray and IGPM. The computation costs of the core functions (*Seed-finder*, *Neighbor-expander* and *Bridge*) are $O(N_V N_{QV})$, $O(kN_V N_{QV})$ and $O(N_E)$ respectively. However, the main performance bottleneck is random walk with restart (RWR) process, the computation cost is at least $O(N_V^2)$ even iterative approximate algorithm is applied.

We also proposed the **IGPM-PEM**, of which incremental graph pattern matching component is also consists of three incremental core functions (Seed-Finder, Neighbor-Expander and Bridge). The re-computation cost is primary determined with the number of vertices for goodness score re-computations. The goodness score is based on the closeness of each vertices computed from the random walk with restart (RWR), and the computation complexity of RWR is more than square of the number of total vertices.

We applied graph clustering approach to reduce the cost of RWR. The computation time of RWR may also be reduced by the implementations such as using GPU, but it is still hard to handle large graphs with million or billion-scale vertices.

Figure 6.1 shows the breakdown of the elapsed time for the original (Batch) G-Ray and the naive version of IGPM (Incremental).

In the *Seed-Finder*, it must re-compute the goodness scores of the updated vertex and its neighbor vertices. *Neighbor-Expander* and *Bridge* functions also must re-compute the goodness scores to find better neighbor vertices and paths than current ones.

The computation costs of IGPM is following. The complexities of incremental RWR, *Seed-*

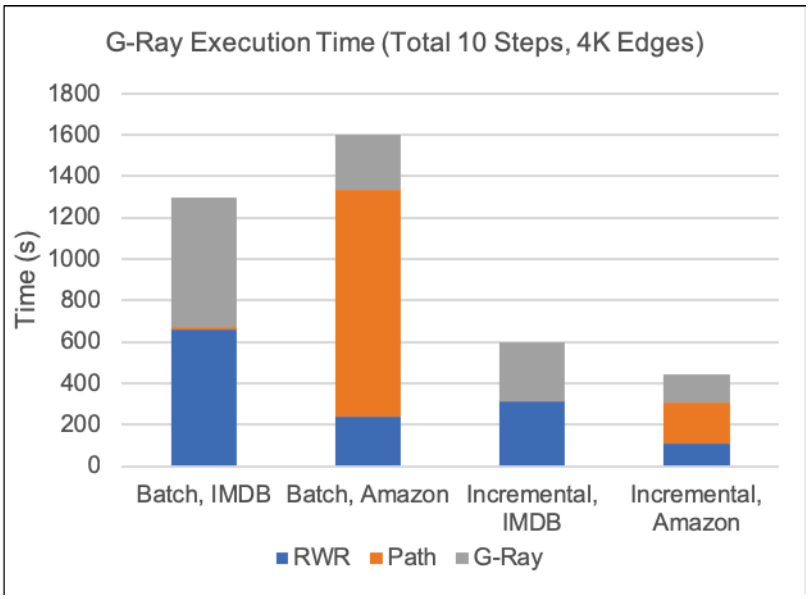


Figure 6.1: Breakdown of the original (Batch) G-Ray and the naive version of IGPM (Incremental) processing for IMDb and Amazon data sets.

finder, incremental *Neighbor-expander* and incremental *Bridge* are $O(n_V(t)N_V)$, $O(n_V(t)N_{QV})$, $O(kn_V(t)N_{QV})$ and $O(N_E)$ respectively. In the incremental version, RWR is still primary performance bottleneck, but the absolute computation time for RWR is much smaller than the original G-Ray. The reason is that the number of vertices for RWR computations becomes much smaller due to the graph clustering.

6.4 Precision of the Parallel and Distributed Traffic Simulation

In the agent-based traffic simulation, we proposed performance optimization methods in adaptive adjustment approaches. One is a dynamic synchronization interval adjustment and the others dynamic agent re-assignment inner and across computation processes.

These methods, however, may cause a little inconsistency of the status of the migrated vehicles. If some simulation processes iterate simulation steps relatively quickly and others do slowly, many vehicles migrated among these processes will run on wrong positions. Even such a partial inconsistency of vehicle positions, it propagates to other areas and leads to unusual

traffic congestion.

Figure 6.2 is an example of the inconsistency in a parallel and distributed traffic simulation. There are two simulation processes running in parallel without sufficient synchronizations. Suppose the current simulation step of the process 1 is T and process 2 is $T + T'$ (the process 2 is running faster than the process 1), and the vehicle V_1 is about to migrate from the process 1 to process 2 at the cross point CP_2. After the migration, V_1 is located at CP_2 in the process 2. However, the CP_2 is the position where V_1 is running at the timestamp T, and it should have run further in the timestamp $T + T'$. It might cause inconsistencies of the vehicle ordering. For example, V_1 runs following V_2 towards CP_3 after this migration, but V_1 should have been in front of the V_2.

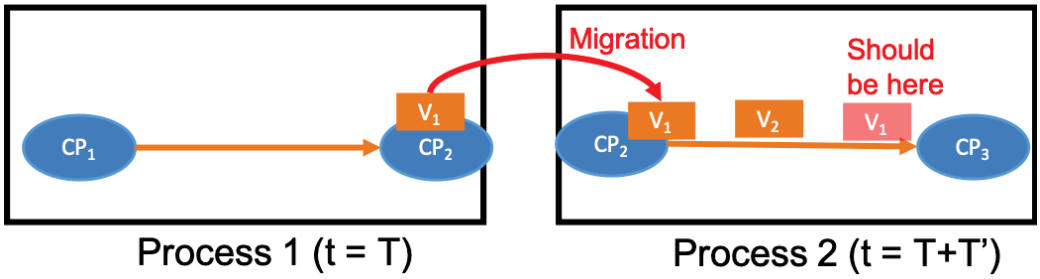


Figure 6.2: An example of the inconsistency due to lack of synchronizations in the parallel and distributed agent-based traffic simulation.

Ajay et, al. proposed several heuristic methods for iterative graph algorithms such as PageRank in [92]. They reduced the computation costs of the iterative processing by the loop perforation and incomplete synchronizations, which are approximate approach and similar to one of our performance optimization method in the traffic simulation.

6.4.1 Evaluation Experiment

In order to inspect the influence of the parallelization and the reduction of synchronization frequency as our optimization method, we conducted additional experiments and evaluated the precision as follows: (1) The number of vehicles with the travel time drastically changed by the number of the road network partitions (X10 processes), (2) The number of vehicles with the travel time drastically changed by the synchronization intervals. The metrics of "precision" means the number (or proportion) of vehicles ran from origin to destination with the near travel time compared to the baseline simulation with a single X10 process (no road

network partitions) and consecutive synchronizations for every simulation steps.

We conducted traffic simulation at TSUBAME 3.0 supercomputer nodes. The specification data of the computation nodes is in Table 6.4.

CPU	Intel Xeon E5-2680 V4 Processor (Broadwell-EP, 14 cores, 2.4GHz) 2 Sockets
Operating Systems	SUSE Linux Enterprise Server 12 SP2
Memory	256GB (DDR4-2400 32GB x 8)
Network	Intel Omni-Path 100 Series 24 Slot Director Class Switch

Table 6.4: TSUBAME 3.0 Node Specifications.

The parameters for the experiment are shown in Table 6.5. Other specifications of the simulation including number of simulation steps, road network and trip data are same as the previous experiments.

Number of X10 Places (computation nodes)	1, 2, 4, 6, 8
Number of X10 Threads	12
Maximum Java Heap Memory	80GB
Minimum Java Heap Memory	160GB
Java Stack Size	64MB
X10 Version	2.5.4
Java Version	Java(TM) SE Runtime Environment (build 1.8.0_181-b13)

Table 6.5: Megaffic/XAXIS Execution Parameters in TSUBAME 3.0.

Figure 6.3 shows the total simulation time with number of computation nodes (X10 Places) and synchronization intervals.

In single node executions, the simulation times with larger synchronization intervals are a little longer than that with synchronizations in every step. In the distributed executions with four or more computation nodes, on the other hand, the simulation time with larger interval becomes faster. Compared to the single-node execution, the simulation time of 6-node execution decreased by 41% in 100 steps/synchronization, while it decreased only by 32% in 1 step/synchronization. As a result, parallel execution with multiple computation nodes and the increase of synchronization interval are effective to optimize the agent-based traffic simulation.

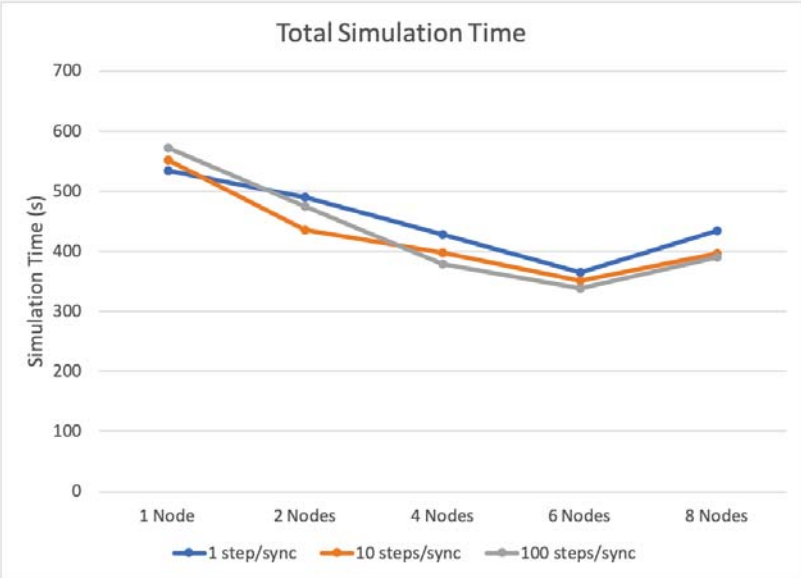


Figure 6.3: Total simulation time with number of nodes and synchronization intervals

There are several reasons why the speed-up ratio is only up to 41% even using six computation nodes. One of the most significant reason is the scale of the simulation: the number of cross points, roads, and running vehicles. In [48], we evaluated the scalability of Megaffic/XAXIS with TSUBAME2.0 supercomputer nodes and road network data sets of Tokyo. With more than 100,000 vehicles, we achieved about eight times speed-up with 16 nodes.

Another reason is the serialization and deserialization costs of Java objects such as vehicle migrations. If we port the Megaffic/XAXIS implementation to X10 and C++ and execute it as a native application, it will reduce these overheads and the effect of scalability will be more significant.

6.4.2 Precision Evaluation with Real Trip Data Set

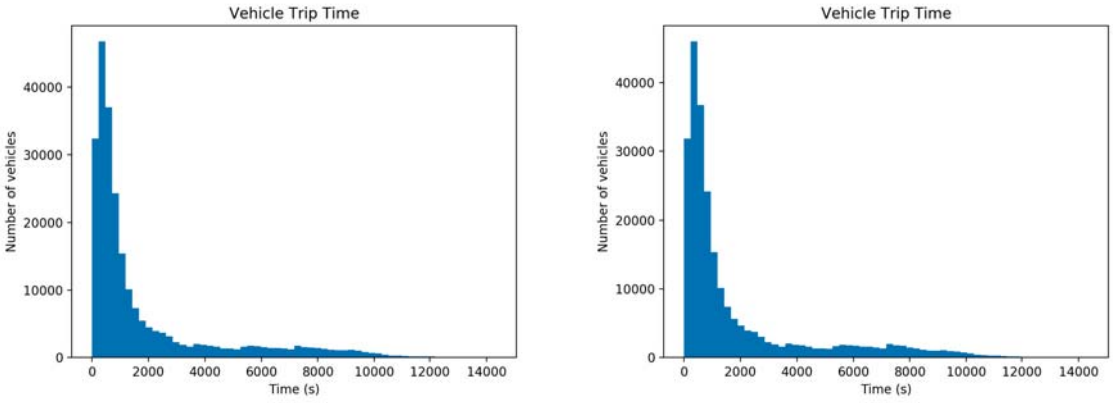
Even the increase of computation nodes and synchronization interval make the traffic simulation faster, the error of simulation results such as travel time might also be increasing. In this section, we evaluate the precision of the simulation outputs quantitatively.

We used the relative error RE of vehicle travel time of the baseline (single node and 1 step/sync) simulation T_b and the optimized simulation T_o results as the measure of this

precision evaluation (Formula 6.8). In [93], we evaluated the precision of multi-modal traffic simulation in Dublin city using the fitness confirmation of vehicle (private cars and Dublin buses) travel time. In [94], the authors used the modeled and observed travel time differences as one of the measures for model calibrations. Their system calibrates traffic model so that more than 85% of vehicles run within 15% error of travel time.

$$RE(\%) = \frac{|T_o - T_b|}{T_b} \times 100 \tag{6.8}$$

Figure 6.4a is the distribution of the travel times (simulation steps) for all vehicles in the simulation with a single computation node and every-step synchronizations. Figure 6.4b is the distribution of the travel times in the fastest simulation with six computation nodes and 100 steps/synchronizations.



(a) Baseline (Single node, 1 step/synchronization) (b) Fastest (6 nodes, 100 steps/synchronization)

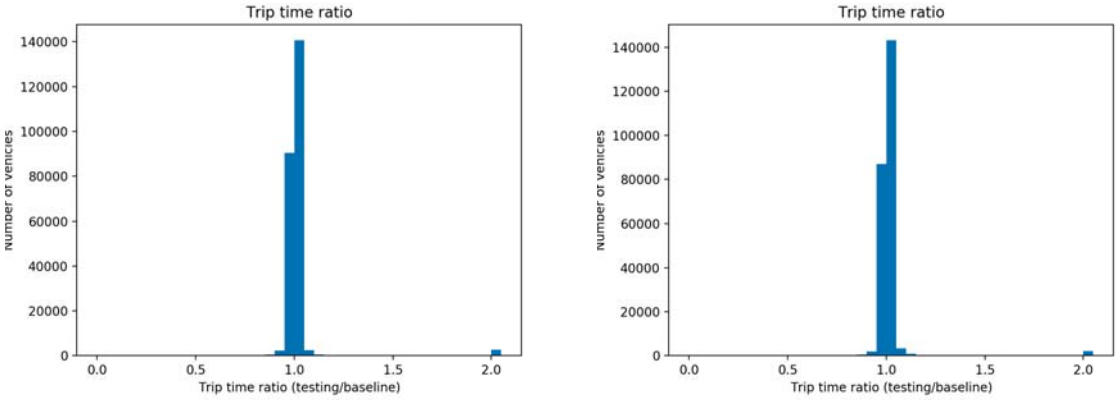
Figure 6.4: Distribution of elapsed travel times for all vehicles

In both simulations, most of the vehicles finished their trips within 3,600 simulation steps, which corresponds to an hour in the real world. Roughly there is little difference between these distributions.

To investigate the difference more microscopically, we compared travel times based on the result of single-node and every-step synchronization for each vehicle and counted the proportion of vehicles with large trip-time differences.

Figure 6.5a and 6.5b are the distributions of the travel time ratio compared to the baseline simulation with 10 steps/synchronization and 100 steps/synchronization respectively. The

horizontal axis indicates the ratio of travel time: the travel time of this simulation divided by that of baseline simulation. The number of vehicles with more than twice travel time than the baseline (more than 2.0) is indicated at the position of 2.0.



(a) Single node, 10 steps/synchronization

(b) Single node, 100 steps/synchronization

Figure 6.5: Distribution of the travel time ratio with single node executions against the baseline simulation.

Figure 6.6 shows the proportions of outlier trips with large travel time difference compared to the baseline simulation. Only 1.2% and 1.0% of vehicles ran for more than twice travel time respectively, and 98% of vehicles are within 10% difference of travel times compared to the baseline simulation.

Figure 6.7a and 6.7b are the distributions of the travel time ratio with 6-node executions. With 6-node simulations, the proportions of the number of vehicles in the with twice or more travel time increased to 2.6%. However, the travel time difference of 93% of vehicles is still within 10%.

Figure 6.8a and 6.8b are the proportions of outlier trips with large travel time difference compared to the baseline simulation (single node execution and every-step synchronizations). The horizontal axis indicates the threshold of the ratio of the travel time difference, and the vertical axis indicates the proportion of the number of total vehicles with outlier travel time. In the simulation 8-node and 10 steps/synchronization execution, for example, 60% of the travel times have more than 2% errors compared to the baseline execution (Figure 6.8a).

In the result of 10 steps/synchronization (Figure 6.8a), even it is also a single-node execution 33% of travel times have more than 1% error compared to the result of the baseline

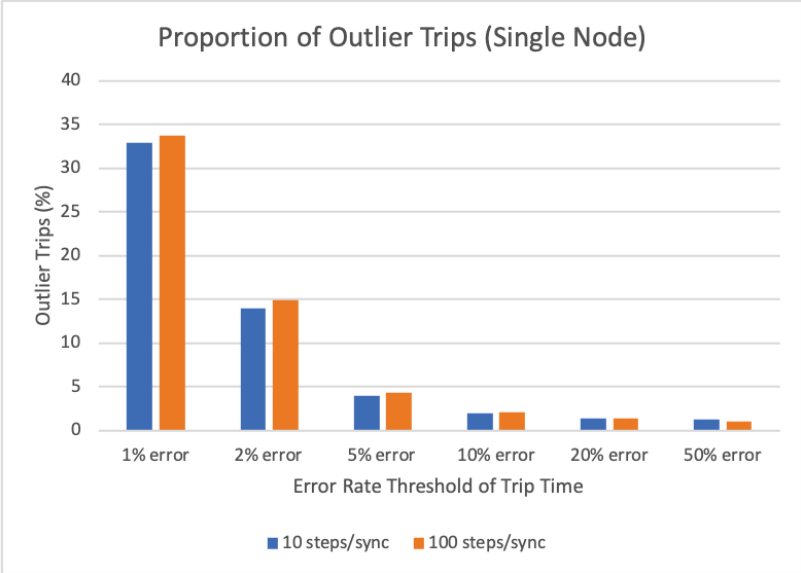
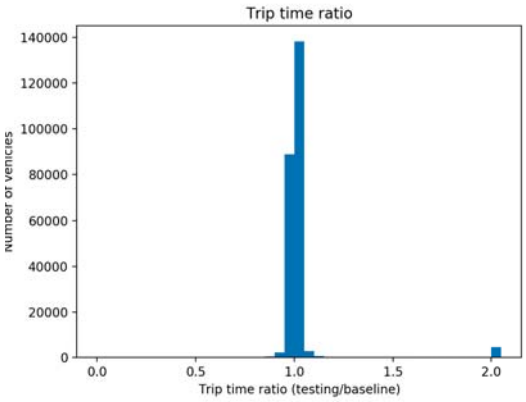
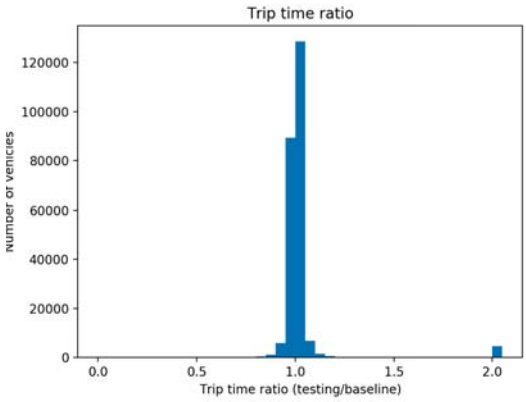


Figure 6.6: Proportion of outlier trips from the single-node and larger synchronization-interval simulations.

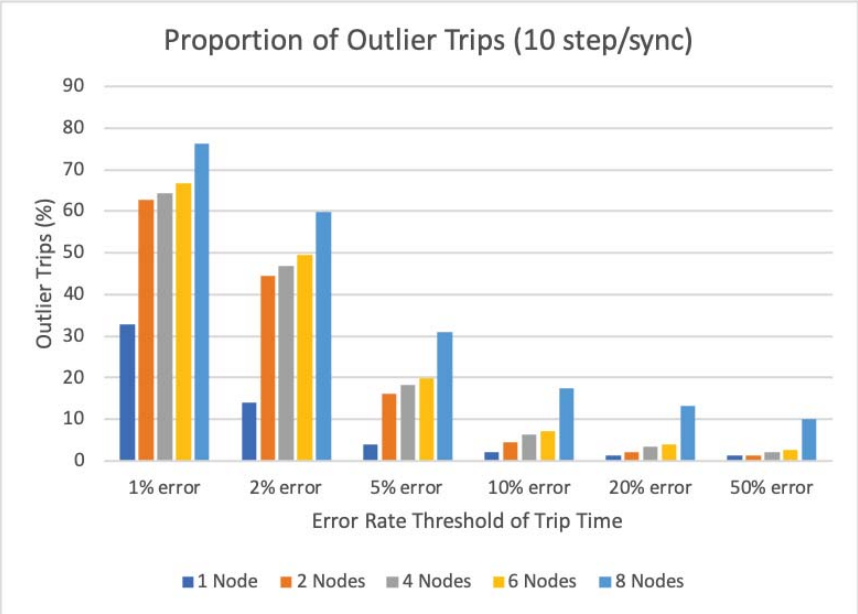


(a) 6 nodes, 10 steps/synchronization

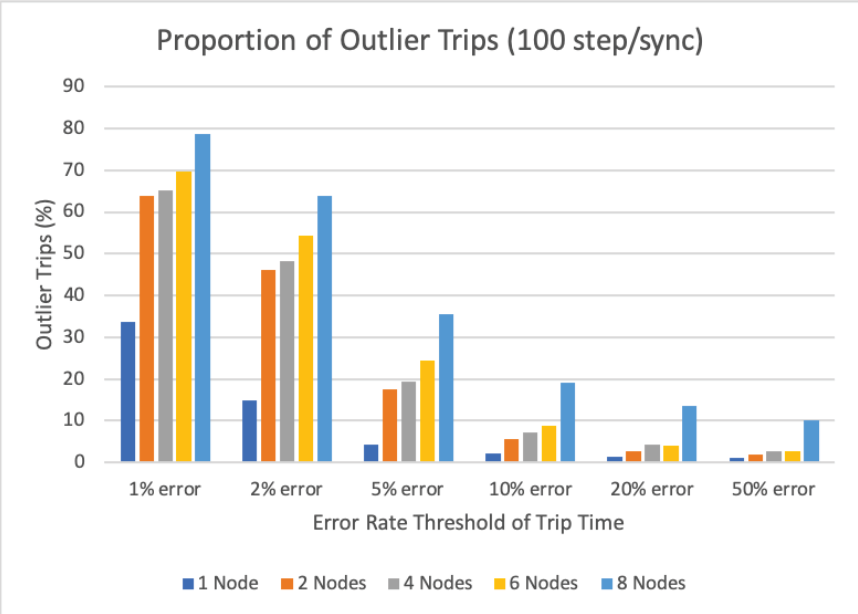


(b) 6 nodes, 100 steps/synchronization

Figure 6.7: Distribution of the travel time ratio with 6-node executions against the baseline simulation.



(a) Proportion of outlier trips with 10 steps/synchronization



(b) Proportion of outlier trips with 100 steps/synchronization

Figure 6.8: Proportion of outlier trips with multiple-node and larger synchronization-interval simulations compared to the baseline simulation.

simulation synchronizes in every step. It is the effect of the synchronization interval adjustment, but the proportion can be regarded as small enough if a higher error rate is allowed. If it allows 5% and 10% errors, the proportions of outlier trips will be 4.0% and 2.0% respectively.

In parallel executions with 2, 4 and 6 nodes, the proportions of outlier trips have a little difference regardless of the error rate thresholds. If it allows only 1% travel time errors, the outlier proportion will be from 63% to 67%, and it will be from 5% to 7% for 10% travel time error allowances. For 1,2,4,6-node executions, moreover, the proportions of outlier trips with more than 50% error are similar (1.2% - 2.6%). That means the number of vehicles mostly affects the multiple nodes are almost constant, and these trips might be stuck in traffic congestion.

The result of 100 steps/synchronization execution (Figure 6.8b), is similar to that of 10 steps/synchronization, but the proportion of outlier trips is up to 5% larger. In particular, the proportions from 6-node execution is 6.8% larger than that of 2-node with the 5% error allowance. That means the synchronization interval also affects the precision for many-node execution, but the difference is much smaller than the difference between single-node and multi-node executions.

To sum it up, the number of computation nodes affects the travel time rather than the synchronization interval. Notably, the result of single-node executions much differs to that of multiple-node execution even it is 2-node.

6.4.3 Evaluation with Random Trip Data Set

Previously we used the real trip data set of commuting vehicles in Dublin city, and showed the limitation of performance with the number of computation nodes and possible error rates. However, the effect of parallel distributed simulaiton with many nodes and the error rates may differ with the data sets. For example, if the number of running vehicles is large, the scalability will be gained. In this section, we evaluated the performance and precision of our proposed methods with a synthetic trip data set.

We generate 100,000 vehicle trips. For each trip, we choose two cross points from the road network data in Dublin city which we used, and then we compute the shortest path between these cross points. In this case, the trip lengths vary according to the chosen cross point set. We perform the traffic simulation with these generated trips. The number of simulation steps is the same (14,000 steps, corresponds four hours).

Figure 6.9 shows the total simulation time with 100,000 synthetic trips. Compared to the

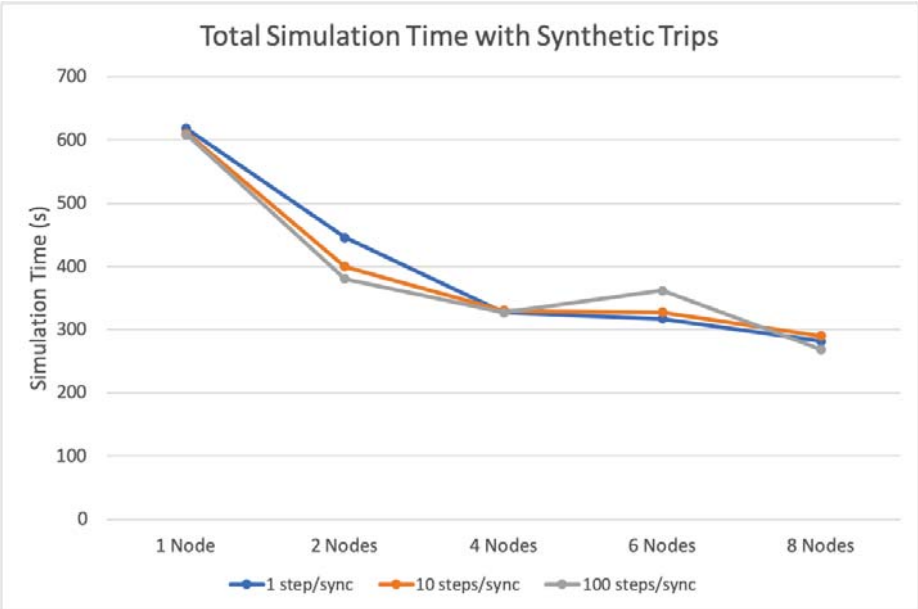


Figure 6.9: Total simulation time with 100,000 synthetic trips.

real trip data, the simulation time has almost constantly decreased to eight nodes. Although the total number of trips is smaller than half of the real trips, the scalability becomes better. The reason is that all vehicles start at the first simulation step, so many vehicles are running simultaneously from the beginning of the simulation. On the other hand, there is no significant effect by larger synchronization intervals. The main reason is that too many vehicles migrated in every step, so the synchronization cost itself is hidden by the communitcation cost by the vehicle migrations.

Figure 6.10 shows the proportions of outlier trips with large travel time differences compared to the baseline simulation (single node execution and every-step synchronizations). The error rates of parallel executions are still around 15% even the error rate threshold is 50%. The high error rates are from traffic congestions and frequent migrations. As a consequence, our optimization method is suitable for low-congestion traffic simulations with large road networks and decentralized trip time, but not for congested traffics with simultaneous many running vehicles.

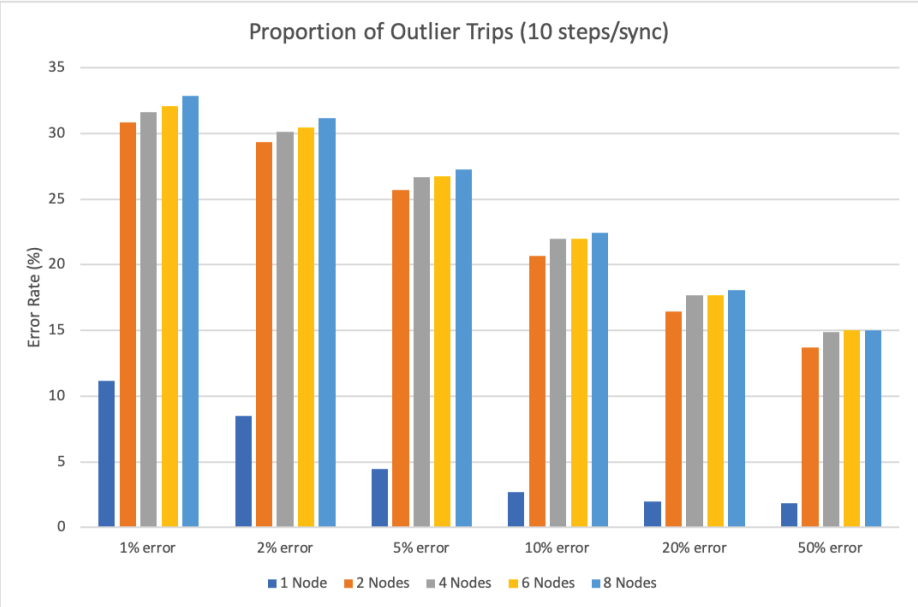


Figure 6.10: Proportion of outlier trip times with multiple computation nodes and synthetic trips and 10 steps per synchronization compared to the baseline simulation.

6.5 Incremental Overlapping Community Detection

In the previous section, we evaluated the performance efficiency of our proposed algorithm. We also evaluate and discuss the precision of our incremental algorithm compared to the original DEMON algorithm in this section.

Our IncrementalDEMON algorithm reduces computation costs by updating local structural changes. However, the result may differ from the original DEMON because of the following reasons.

1. When only part of local vertices and edges are added, the IncrementalDEMON outputs very large communities because the added vertices have the same labels by the incremental label propagation.

2. Even some edges are removed and large communities are divided, the output communities as the result of IncrementalDEMON might not be divided.

6.5.1 Precision Evaluation of IncrementalDEMON with Real Data Set

To evaluate the effect of these incremental updates, we conducted additional experiments. First, we applied the original DEMON to the social networks and outputted the distribution of output community size. Figure 6.11 describes the community size distribution of a subgraph of the Amazon data set with 100,000 vertices.

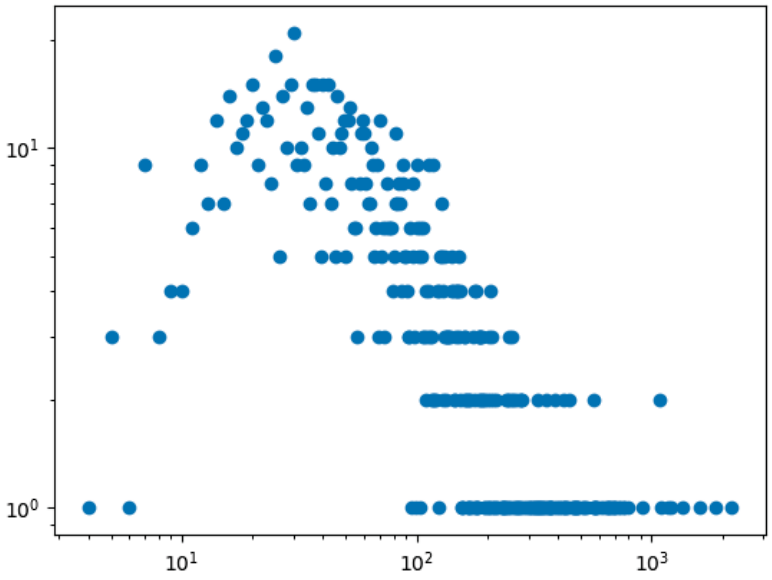


Figure 6.11: Distribution of community size by the original DEMON algorithm with the Amazon data set. The horizontal axis and vertical axis indicate the community size and the number of communities respectively.

Next, we applied IncrementalDEMON processes to the subgraph removing the latest 1,000 vertices and then restoring the removed 1,000 vertices and connecting edges. Figure 6.12 shows the difference of community size distributions. After applying IncrementalDEMON (red dots), the number of communities with 10-100 vertices are drastically different from before incremental processes (green dots).

Table 6.6 shows the statistical features of the number of communities and community size as a result of a dense graph (Amazon data set). "Before" means the result of the original DEMON algorithm with 100,000 vertices and "After" means the result of the IncrementalDE-

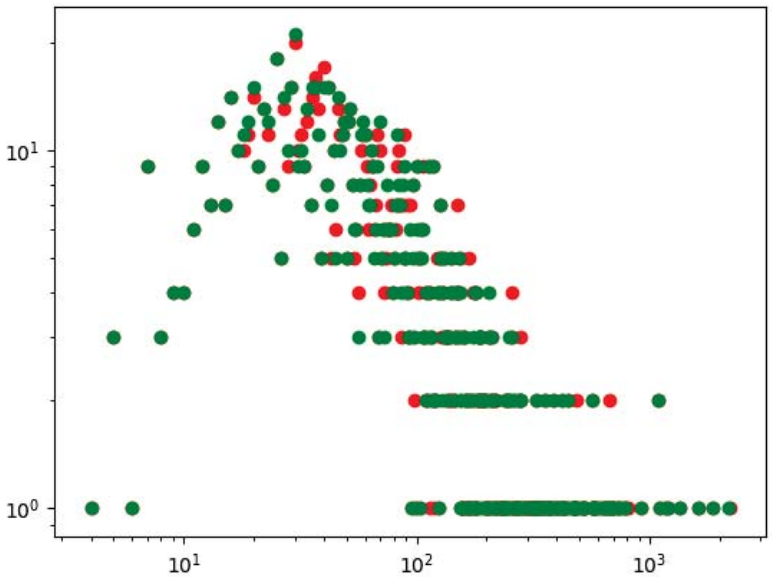


Figure 6.12: Distribution of community size after the incremental DEMON algorithm with removing 1,000 vertices and restoring these vertices and edges. The green and red dots indicate the results before and after applying IncrementalDEMON respectively.

MON algorithm removing and restoring 1,000 vertices and connecting edges. The parameter ϵ is the threshold for the Merge function of DEMON algorithm.

	$\epsilon = 0.2$		$\epsilon = 0.5$		$\epsilon = 0.8$	
	Before	After	Before	After	Before	After
Number of communities	103	103	28	28	6	6
Maximum size	497	538	841	841	1547	1823
Minimum size	4	4	25	25	327	327
Average size	88.4	91.0	269.3	309.5	980.7	1091.0
Median size	49.0	49.0	272.5	279.0	1040.5	1128.5
Standard deviation of size	106.3	110.6	175.6	201.3	465.2	582.9

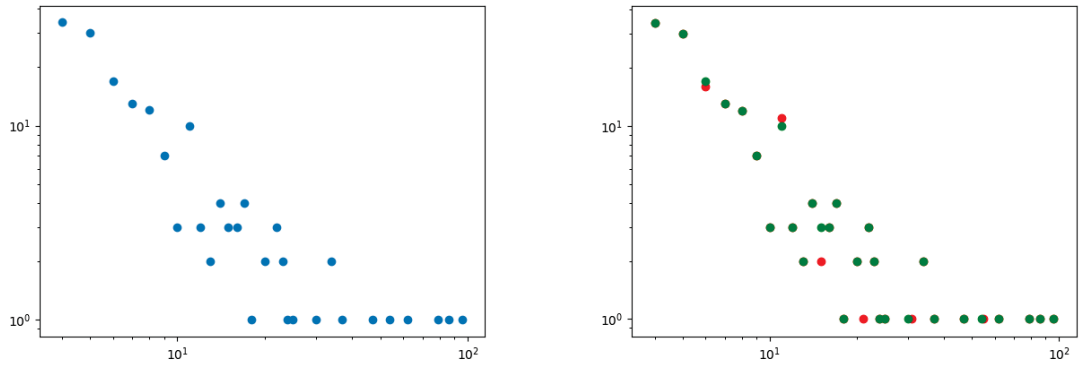
Table 6.6: Statistical features of communities of the original DEMON and IncrementalDEMON with Amazon data set.

With the all parameter values ϵ for *Merge* function, the standard deviation of community size increase after applying IncrementalDEMON, particularly for larger parameter value even the maximum and minimum sizes of communities do not change drastically. The reason is that newly added vertices in our IncrementalDEMON algorithm are likely to join large communities due to the implementation of *LabelPropagation* function, and some vertices moved from smaller communities to larger neighboring communities.

In the Figure 6.13a and 6.13 describes the community size distribution by the original DEMON algorithm and IncrementalDEMON algorithm with 10,000 vertices of the IMDb data set.

Table 6.7 shows the statistical features of the number of communities and community size as a result of a sparse graph (IMDb data set). Because the number of vertices is relatively small, the initial number of vertices of the subgraph is 10,000. For the IMDb data set, the distribution of community size does not drastically change before and after applying IncrementalDEMON process. The reason is that most of the removed and restored vertices are not connect to many other vertices which belong to large communities and rejoin to neighbor small communities after restoration.

Our IncrementalDEMON algorithm is suitable for sparse graphs to provide similar results as the original DEMON algorithm. For dense graphs, on the other hand, it is likely to generate a few unusually large communities and many small communities.



(a) Community size distribution by the original DEMON algorithm

(b) Community size distribution by the IncrementalDEMON algorithm with removing/restoring 1,000 vertices.

Figure 6.13: Distribution of community size by the original DEMON and IncrementalDEMON with IMDb data set.

	$\epsilon = 0.2$		$\epsilon = 0.5$		$\epsilon = 0.8$	
	Before	After	Before	After	Before	After
Number of communities	165	165	130	129	110	110
Maximum size	96	96	110	114	198	198
Minimum size	4	4	4	4	4	4
Average size	11.1	11.2	12.9	13.1	13.1	13.4
Median size	7.0	7.0	7.0	7.0	6.5	6.5
Standard deviation of size	13.6	13.6	17.5	17.8	23.1	23.3

Table 6.7: Statistical features of communities of the original DEMON and IncrementalDEMON with IMDb data set.

6.5.2 Precision Evaluation of IncrementalDEMON with Synthetic Data Set

We found that the scale, number of added vertices and edges, and density of graphs will affect the output precision. In order to evaluate the influence of the incremental process more quantitatively, we conducted another experient with synthetic data.

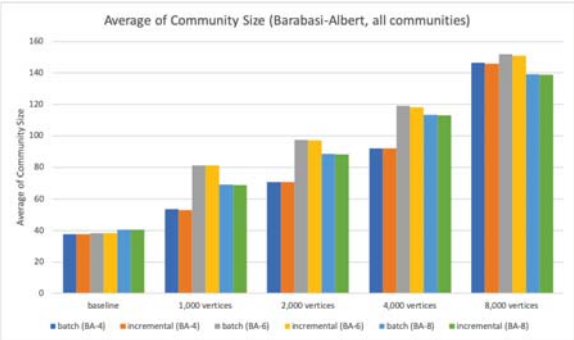
First, we generated two types of synthetic grah data. First is "Barabasi-Albert Graph" [95], which has scale-free and time-evolving features and such kind of graphs can be found at WWW and citation networks. The second is "Powerlaw-Cluster Graph" [96], which is similar to the Barabasi-Albert graph, but it has high clustering and local transitivity to describe the follower-follwing relationships in social networks more accurately. We generated the following data sets (Table 6.8) with NetworkX random graph generator. We set the number of vertices of the base graph to 10,000, and then we generated larger graphs with 18,000 vertices to simulate the network growths.

Graph Name	Graph Type	Number of Vertices	Average Degree
BA-4	Barabasi-Albert	10,000	4
BA-6	Barabasi-Albert	10,000	6
BA-8	Barabasi-Albert	10,000	8
PC-4	Powerlaw-Cluster	10,000	4
PC-6	Powerlaw-Cluster	10,000	6
PC-8	Powerlaw-Cluster	10,000	8

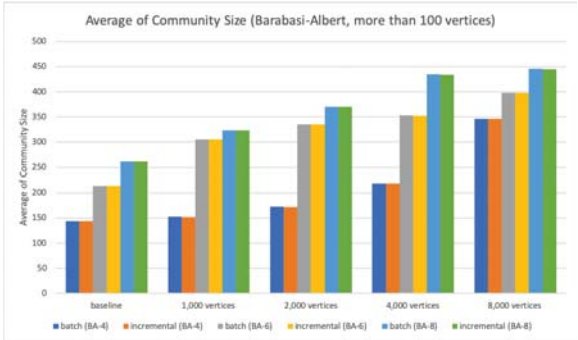
Table 6.8: Synthetic graph data set.

We performed the batch version of DEMON and our IncrementalDEMON algorithms with adding 1,000, 2,000, 4,000 and 8,000 vertices and connecting edges to the base graph.

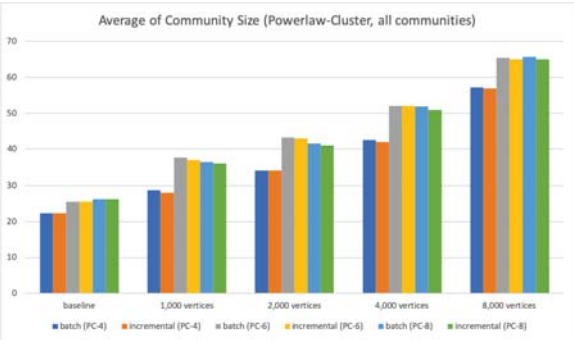
Figures 6.14 are the result of community size. In the all data sets, the difference of community size between the batch and incremental algorithms is within one vertices. The trend of the community size difference is still same for larger communities with more then 100 vertices in the Figure 6.14b and Figure 6.14d. As a consequence, the IncrementalDEMON is suitable for sparse scale-free graphs with lower average degrees.



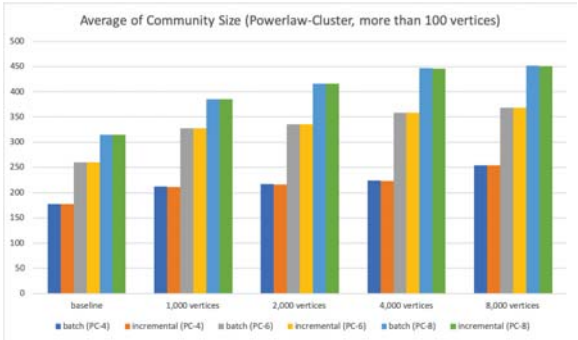
(a) Barabasi-Albert Graph (all communities)



(b) Barabasi-Albert Graph (large communities)



(c) Powerlaw-Cluster Graph (all communities)



(d) Powerlaw-Cluster Graph (large communities)

Figure 6.14: Average community size of batch DEMON and IncrementalDEMON with synthetic graph data

6.6 Data Size and Implementation

We used Megaffic as the traffic simulation application written in Java, and implemented XAXIS as the simulation base in X10 (Java back-end). We also implemented IGPM-PEM in Python with its packages such as NetworkX and Keras. The main reason we used Java and Python instead of C and C++ is to reduce engineering and implementation costs.

Our implementation and environment can handle only up to million-scale graphs, and we plan to port these implementations to lower-level programming languages such as C and C++ and parallel programming libraries and devices such as MPI and GPU to manage larger graphs in parallel. These implementations of porting and parallelization are not directly related to our proposed methods, and we will work on them in the future.

Chapter 7

Conclusion

7.1 Overall Conclusion

Large-scale graph analytics and agent-based simulations are widely used at many real-world applications. In the real-world applications, time-evolving and large-scale data represented as a large graph with million and billion vertices and edges. Various graph algorithms have been proposed with the demands of feature extractions from real-world graph data, and optimization methods of these graph algorithms have also been proposed.

Although many performance optimization methods for graph analytics algorithms and agent-based simulations have been proposed, these methods did not yet catch up with the growth of real-world networks. Moreover, many graph algorithms have limitations of input graphs and output results. For example, some graph pattern matching algorithms do not guarantee topological matching with the query pattern graph, and many community detection algorithms do not allow overlapping communities. Such limitations prevent us from applying these algorithms to time-evolving and billion-scale complex networks.

To address these performance problems, we proposed performance optimization methods applicable to time-evolving and large-scale real-world graphs; adaptive object re-assignments and synchronization interval adjustment for agent-based traffic simulations in parallel and distributed computing environments, incremental community detection algorithm named **IncrementalDEMON**, and adaptive and incremental graph pattern matching algorithm named **IGPM-PEM**.

In the agent-based traffic simulation, we investigated the prime performance bottleneck

of the simulation framework named XAXIS and traffic application Megaffic at first. With the result of the investigation, we proposed three types of adaptive performance optimization methods to gain the scalability; synchronization interval adjustments, agent re-assignments within each computation process and agent re-assignments in among computation processes. The adaptive synchronization interval adjustments method achieved up to 35% speed-up with 16 computation nodes. In the dynamic agent re-assignment methods, the re-assignments in among processes reduced the total computation time.

IncrementalDEMON updates the overlapping communities incrementally with the additional and removals of vertices and edges. It updates and merges small communities incrementally in the minimum frequency. With our proposed incremental processing, we achieved up to 101 times speed-up compared to the original community detection algorithm.

IGPM-PEM is an adaptive, approximate and incremental pattern matching algorithm based on an approximate subgraph isomorphism algorithm named G-Ray, and consists of two components; incremental graph pattern matching (IGPM) to find other matching patterns incrementally and partial execution manager (PEM) to adjust the number of vertices for recomputations adaptively. The IGPM achieved up to 10.1 times speed-up compared to the original G-Ray algorithm, and adaptive IGPM with PEM gained the speed-up by up to 1.17 times.

The series of researches, proposals shows some guides and sample solutions to the problem how can we analyze the real-world graph data efficiently with the parallel and distributed computing environment such as supercomputers. The discussion of our proposals also shows the possibility of a broader range of real-world applications with graph analytics.

7.2 Future Work

As future work, we still have some research issues related to our proposed methods. The following work will be further directions to enhance the possibilities of more extensive applications of our proposals.

First, we need more discussion and verification about the generalization of our proposed methods for incremental graph processing. In our IGPM-PEM method, we proposed the partial execution method framework specialize to our incremental graph pattern matching, so we need to generalize the framework for other incremental graph analytics such as community detection algorithms. Moreover, our proposed adaptive optimization methods are also spe-

cialized our traffic applications (Megaffic and XAXIS). Based on our series of discussions in Chapter 6, we will improve our processing models and the implementation of our proposed algorithms, and then verify the efficiency and applicability in other algorithm and simulation frameworks.

Second, the implementations and performance evaluations of our proposed algorithms have many issues to be improved. Although our incremental and adaptive algorithms are intended to be used at parallel and distributed computing environments with million- and billion-scale graph data, we still cannot make full use of such environments because we mainly implemented them in Python and Java. In the future, we will re-implement them to C/C++ with parallel processing libraries and devices such as OpenMP, MPI and GPU, and re-evaluate the effectiveness of our methods again. Our current evaluation results have limitations due to Java and Python object management such as serializations and deserializations, so the re-implementations will result to show more remarkable scalability with our proposals.

With these remaining works of generalization and re-implementations, we will apply these applications to other real-world applications with more massive data sets.

References

[1] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[2] Jure Leskovec, Lada A Adamic, and Bernardo A Huberman. The dynamics of viral marketing. *ACM Transactions on the Web (TWEB)*, 1(1):5, 2007.

[3] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Predicting positive and negative links in online social networks. In *Proceedings of the 19th international conference on World wide web*, pages 641–650. ACM, 2010.

[4] Masaru Watanabe and Toyotaro Suzumura. How social network is evolving?: a preliminary study on billion-scale twitter network. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 531–534. ACM, 2013.

[5] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. AcM, 2010.

[6] Hanghang Tong, Christos Faloutsos, Brian Gallagher, and Tina Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 737–746. ACM, 2007.

[7] T Osogami, T Imamichi, H Mizuta, T Morimura, R Raymond, T Suzumura, R Takahashi, and T Ide. Ibm mega traffic simulator. Technical report, IBM Research Report, RT0896, 2012.

[8] Edgar Alonso Lopez-Rojas and Stefan Axelsson. Social simulation of commercial and financial behaviour for fraud detection research. In *Social Simulation Conference. Bellaterra, Cerdanyola del Valles, 1a: 2014*, 2014.

[9] George Karypis and Vipin Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359 – 392, 1999.

[10] Michele Coscia, Giulio Rossetti, Fosca Giannotti, and Dino Pedreschi. Demon: a local-first discovery method for overlapping communities. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 615–623. ACM, 2012.

[11] Albert-László Barabási, Réka Albert, and Hawoong Jeong. Scale-free characteristics of random networks: the topology of the world-wide web. *Physica A: statistical mechanics and its applications*, 281(1-4):69–77, 2000.

[12] Leman Akoglu, Hanghang Tong, and Danai Koutra. Graph based anomaly detection and description: a survey. *Data mining and knowledge discovery*, 29(3):626–688, 2015.

[13] Stephen Ranshous, Shitian Shen, Danai Koutra, Steve Harenberg, Christos Faloutsos, and Nagiza F Samatova. Anomaly detection in dynamic networks: a survey. *Wiley Interdisciplinary Reviews: Computational Statistics*, 7(3):223–247, 2015.

[14] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[15] Srinivasan Parthasarathy, Shirish Tatikonda, and Duygu Ucar. A survey of graph mining techniques for biological datasets. In *Managing and mining graph data*, pages 547–580. Springer, 2010.

[16] Shijie Zhang, Shirong Li, and Jiong Yang. Gaddi: distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 192–203. ACM, 2009.

[17] Albert-László Barabási et al. *Network science*. Cambridge university press, 2016.

[18] P Symeon, K Yiannis, V Athena, and S Ploutarchos. Community detection in social media, performance and application considerations. *Journal of Data Mining Knowledge Discovery*, 24(3):515–554, 2012.

[19] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[20] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[21] Mark EJ Newman. Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582, 2006.

[22] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. Maximizing modularity is hard. *arXiv preprint physics/0608255*, 2006.

[23] Thang N Dinh and My T Thai. Community detection in scale-free networks: approximation algorithms for maximizing modularity. *IEEE Journal on Selected Areas in Communications*, 31(6):997–1006, 2013.

[24] Vincent A Traag. Faster unfolding of communities: Speeding up the louvain algorithm. *Physical Review E*, 92(3):032801, 2015.

[25] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 76(3):036106, 2007.

[26] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[27] Joao P Hespanha. An efficient matlab algorithm for graph partitioning. *Santa Barbara, CA, USA: University of California*, 2004.

[28] Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.

[29] Steve Gregory. Finding overlapping communities in networks by label propagation. *New Journal of Physics*, 12(10):103018, 2010.

[30] Anita Zakrzewska and David A Bader. Fast incremental community detection on dynamic graphs. In *International Conference on Parallel Processing and Applied Mathematics*, pages 207–217. Springer, 2015.

[31] Weiyi Liu, Toyotaro Suzumura, Lingli Chen, and Guangmin Hu. A generalized incremental bottom-up community detection framework for highly dynamic graphs. In *Big Data (Big Data), 2017 IEEE International Conference on*, pages 3342–3351. IEEE, 2017.

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

[33] Robin Milner. *Communication and concurrency*, volume 84. Prentice hall New York etc., 1989.

[34] Monika Rauch Henzinger, Thomas A Henzinger, and Peter W Kopke. Computing simulations on finite and infinite graphs. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 453–462. IEEE, 1995.

[35] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. Graph pattern matching: from intractable to polynomial time. *Proceedings of the VLDB Endowment*, 3(1-2):264–275, 2010.

[36] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. Capturing topology in graph pattern matching. *Proceedings of the VLDB Endowment*, 5(4):310–321, 2011.

[37] Wenfei Fan. Graph pattern matching revised for social network analysis. In *Proceedings of the 15th International Conference on Database Theory*, pages 8–21. ACM, 2012.

[38] Jyun-Sheng Kao and Jerry Chou. Distributed incremental pattern matching on streaming graphs. In *Proceedings of the ACM Workshop on High Performance Graph Processing*, pages 43–50. ACM, 2016.

[39] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, Xin Wang, and Yinghui Wu. Incremental graph pattern matching. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 925–936. ACM, 2011.

[40] Wenfei Fan, Chunming Hu, and Chao Tian. Incremental graph computations: Doable and undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 155–169. ACM, 2017.

[41] Gennaro Cordasco, Rosario De Chiara, Ada Mancuso, Dario Mazzeo, Vittorio Scarano, and Carmine Spagnuolo. Bringing together efficiency and effectiveness in distributed simulations: The experience with d-mason. *Simulation*, pages 1236–1253, 2013.

[42] Gennaro Cordasco, Francesco Milone, Carmine Spagnuolo, and Luca Vicidomini. Exploiting d-mason on parallel platforms: A novel communication strategy. In *European Conference on Parallel Processing*, pages 407–417. Springer, 2014.

[43] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527, 2005.

[44] Anthony Ventresque Quentin Bragard and Liam Murphy. dsumo: Towards a distributed sumo. In *The first SUMO User Conference (SUMO2013)*, 2013.

[45] Daniel Krajzewicz, Georg Hertkorn, C. Rssel, and Wagner Peter. Sumo (simulation of urban mobility) - an open-source traffic simulation. In *Proceedings of the 4th Middle East Symposium on Simulation and Modelling (MESM20002)*, 2002.

[46] Nicolas Lefebvre and Michael Balmer. Fast shortest path computation in time-dependent traffic networks. *Arbeitsberichte Verkehrs-und Raumplanung*, 439, 2007.

[47] José L Galán-García, Gabriel Aguilera-Venegas, María Á Galán-García, and Pedro Rodríguez-Cielos. A new probabilistic extension of dijkstras algorithm to simulate more realistic traffic flow in a smart city. *Applied mathematics and Computation*, 267:780–789, 2015.

[48] Toyotaro Suzumura and Hiroki Kanezashi. Highly scalable x10-based agent simulation platform and its application to large-scale traffic simulation. In *Distributed Simulation and Real Time Applications (DS-RT), 2012 IEEE/ACM 16th International Symposium on*, pages 243–250. IEEE, 2012.

[49] Anthony Ventresque, Quentin Bragard, Elvis S Liu, Dawid Nowak, Liam Murphy, Georgios Theodoropoulos, and Qi Liu. Spartsim: A space partitioning guided by road network for distributed traffic simulations. In *Distributed Simulation and Real Time Applications (DS-RT), 2012 IEEE/ACM 16th International Symposium on*, pages 202–209. IEEE, 2012.

[50] T. Suzumura and H. Kanezashi. Highly scalable x10-based agent simulation platform and its application to large-scale traffic simulation. In *Distributed Simulation and Real*

Time Applications (DS-RT), 2012 IEEE/ACM 16th International Symposium on, pages 243–250, 2012.

[51] T. Suzumura and H. Kanezashi. Accelerating large-scale distributed traffic simulation with adaptive synchronization method. In *20th ITS World Congress 2013*, 2013.

[52] T. Suzumura and H. Kanezashi. A holistic architecture for super real-time multiagent simulation platform. In *Simulation Conference (WSC), Proceedings of the 2013 Winter*, 2013.

[53] Gaku Yamamoto, Hideki Tai, and Hideyuki Mizuta. A platform for massive agent-based simulation and its evaluation. In Nadeem Jamali, Paul Scerri, and Toshiharu Sugawara, editors, *Massively Multi-Agent Technology*, volume 5043 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 2008.

[54] Mordechai (Muki) Haklay and Patrick Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, October 2008.

[55] Google. What is gtfs? - transit - google developers.

[56] Andreas Horni, Kai Nagel, and Kay W Axhausen. *The multi-agent transport simulation MATSim*. Ubiquity Press London:, 2016.

[57] Michael Balmer Rashid A. Waraich, David Charypar and Kay W. Axhausen. Performance improvements for large scale traffic simulation in matsim. In *9th Swiss Transport Research Conference*, 2009.

[58] Martin Rosvall and Carl T Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, 105(4):1118–1123, 2008.

[59] Yong-Yeol Ahn, James P Bagrow, and Sune Lehmann. Link communities reveal multiscale complexity in networks. *Nature*, 466(7307):761–764, 2010.

[60] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

[61] Pascal Pons and Matthieu Latapy. Computing communities in large networks using random walks. In *Computer and Information Sciences-ISCIS 2005*, pages 284–293. Springer, 2005.

[62] Gabriel Tanase, Toyotaro Suzumura, Jinho Lee, Jason Crawford, Hiroki Kanezashi, Song Zhang, Warut D Vijitbenjaronk, et al. System g distributed graph database. *arXiv preprint arXiv:1802.03057*, 2018.

[63] Michele Coscia. Michele Coscia — DEMON, 2012.

[64] Zhenyu Wu and Ming Zou. An incremental community detection method for social tagging systems using locality-sensitive hashing. *Neural Networks*, 58:14–28, 2014.

[65] Sheng Pang, Changjia Chen, and Ting Wei. A realtime community detection algorithm: incremental label propagation. In *Future Information Networks, 2009. ICFIN 2009. First International Conference on*, pages 313–317. IEEE, 2009.

[66] Jingyong Li, Lan Huang, Tian Bai, Zhe Wang, and Hongsheng Chen. Cdbia: a dynamic community detection method based on incremental analysis. In *Systems and Informatics (ICSAI), 2012 International Conference on*, pages 2224–2228. IEEE, 2012.

[67] Krzysztof Michalak and Jerzy Korczak. Graph mining approach to suspicious transaction detection. In *Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on*, pages 69–75. IEEE, 2011.

[68] Harry R Lewis. Computers and intractability. a guide to the theory of np-completeness, 1983.

[69] Ehab Abdelhamid, Mustafa Canim, Mohammad Sadoghi, Bishwaranjan Bhattacharjee, Yuan-Chi Chang, and Panos Kalnis. Incremental frequent subgraph mining on large evolving graphs. *IEEE Transactions on Knowledge and Data Engineering*, 29(12):2710–2723, 2017.

[70] Charith Wickramaarachchi, Rajgopal Kannan, Charalampos Chelmis, and Viktor K Prasanna. Distributed exact subgraph matching in small diameter dynamic graphs. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 3360–3369. IEEE, 2016.

[71] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 337–348. ACM, 2013.

[72] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[73] Jia Li, Yang Cao, and Shuai Ma. Relaxing graph pattern matching with explanations. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 1677–1686. ACM, 2017.

[74] Adam Stotz, Rakesh Nagi, and Moises Sudit. Incremental graph matching for situation awareness. In *Information Fusion, 2009. FUSION'09. 12th International Conference on*, pages 452–459. IEEE, 2009.

[75] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment*, 5(9):788–799, 2012.

[76] Wenfei Fan, Xin Wang, and Yinghui Wu. Querying big graphs within bounded resources. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 301–312. ACM, 2014.

[77] Muhammad Anis Uddin Nasir, Aristides Gionis, Gianmarco De Francisci Morales, and Sarunas Girdzijauskas. Fully dynamic algorithm for top-k densest subgraphs. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 1817–1826. ACM, 2017.

[78] Sutanay Choudhury, Lawrence Holder, John Feo, and George Chin. Fast search for dynamic multi-relational graphs. In *Proceedings of the Workshop on Dynamic Networks Management and Mining*, pages 1–8. ACM, 2013.

[79] Shengqi Yang, Fangqiu Han, Yinghui Wu, and Xifeng Yan. Fast top-k search in knowledge graphs. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 990–1001. IEEE, 2016.

[80] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. Query preserving graph compression. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 157–168. ACM, 2012.

[81] Hanghang Tong and Christos Faloutsos. Center-piece subgraphs: problem definition and fast solutions. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 404–413. ACM, 2006.

[82] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[83] Matthias Plappert. keras-rl. `https://github.com/keras-rl/keras-rl`, 2016.

[84] Jaroslaw Jankowski, Radosaw Michalski, and Piotr Brdka. Spreading processes in multilayer complex network within virtual world, 2017.

[85] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[86] Benjamin Schiller, Jeronimo Castrillon, and Thorsten Strufe. Efficient data structures for dynamic graph analysis. In *2015 11th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS)*, pages 497–504. IEEE, 2015.

[87] Benjamin Schiller, Clemens Deusser, Jeronimo Castrillon, and Thorsten Strufe. Compile- and run-time approaches for the selection of efficient data structures for dynamic graph analysis. *Applied Network Science*, 1(1):9, 2016.

[88] Biagio Cosenza, Gennaro Cordasco, Rosario De Chiara, and Vittorio Scarano. Distributed load balancing for parallel agent-based simulations. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 62–69. IEEE, 2011.

[89] Masatoshi Hanai, Toyotaro Suzumura, Anthony Ventresque, and Kazuyuki Shudo. Towards a framework for adaptive resource provisioning in large-scale distributed agent-based simulation. In *European Conference on Parallel Processing*, pages 430–439. Springer, 2014.

[90] Masatoshi Hanai, Toyotaro Suzumura, Anthony Ventresque, and Kazuyuki Shudo. An adaptive vm provisioning method for large-scale agent-based traffic simulations on the cloud. In *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, pages 130–137. IEEE, 2014.

[91] Guanfeng Liu, Kai Zheng, Yan Wang, Mehmet A Orgun, An Liu, Lei Zhao, and Xiaofang Zhou. Multi-constrained graph pattern matching in large-scale contextual social graphs. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 351–362. IEEE, 2015.

[92] Ajay Panyala, Omer Subasi, Mahantesh Halappanavar, Ananth Kalyanaraman, Daniel Chavarria-Miranda, and Sriram Krishnamoorthy. Approximate computing techniques for iterative graph algorithms. In *High Performance Computing (HiPC), 2017 IEEE 24th International Conference on*, pages 23–32. IEEE, 2017.

[93] Toyotaro Suzumura, Gavin McArdle, and Hiroki Kanezashi. A high performance multimodal traffic simulation platform and its case study with the dublin city. In *Proceedings of the 2015 Winter Simulation Conference*, pages 767–778. IEEE Press, 2015.

[94] Francois Dion, Karthik Sivakumaran, and Xuegang Jeff Ban. Evaluation of traffic simulation model use in the development of corridor system management plans (csmps). Technical report, PATH Research Report, University of California, Berkeley, 2012.

[95] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[96] Petter Holme and Beom Jun Kim. Growing scale-free networks with tunable clustering. *Physical review E*, 65(2):026107, 2002.