International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Design and Development of Parallel Algorithm for Graph Isomorphism

[1]Dr. Vijaya Parag Balpande, [2]Dr. Ujjwala Bal Aher, [3]Dr. Shyam Deshmukh, [4]Rohit Pawar, [5]Pramod B. Dhamdhere, [6]Nilesh Kulal

Submitted: 24/10/2023 **Revised**: 16/12/2023 **Accepted**: 25/12/2023

Abstract: A fundamental issue in mathematics and computer science, the problem of graph isomorphism (GI) has applications in many fields, including chemistry, network analysis, and cryptography. To perform GI, two provided graphs must be examined to see if they are isomorphic, which means they share the same basic structure despite any differences in vertex and edge names. Computational complexity theory has long sought to create effective GI algorithms. Due to the increased accessibility of high-performance parallel computing platforms, there has been an increase in interest in parallel methods for GI in recent years. In order to speed up graph isomorphism testing, parallel methods are created to take advantage of multi-core computers, GPUs, and distributed computing clusters. The goal of this research is to create a parallel GI method that takes advantage of parallel processing to speed up computations for extensive graph comparisons. The algorithm uses a divide-and-conquer tactic, breaking down the input graphs into smaller sub-graphs that are then independently examined for isomorphism. The efficiency of these sub-graph comparisons is greatly increased by running them concurrently across numerous processing units. Load balancing algorithms are incorporated into the algorithm to provide scalability, dividing the workload equally among processing units and reducing communication cost. In order to further improve performance, the algorithm also uses optimisation techniques like graph pruning and canonization. The algorithm's success in lowering the temporal complexity of GI for big graphs is demonstrated by experimental data. This research advances graph isomorphism algorithms by utilising parallelism, which has implications for effectively resolving practical issues and overcoming computational difficulties in a variety of scientific and industrial applications.

Keywords: *Parallel Algorithm, Recursion, Graph Isomorphism, Sub graph, Large Graph*

1. Introduction

A well-known and essential subject in computer science, graph theory, and several other domains is the problem of graph isomorphism (GI). Checking for isomorphism, or if two provided graphs have the same underlying structure and have their vertices and edges mapped in a fashion that respects adjacency connections, entails comparing two given graphs. The Graph Isomorphism problem has applications in a variety of fields, including chemistry, network analysis, data mining, and pattern recognition. It is theoretically fascinating and practically significant [1]. The concept of Graph Isomorphism poses an intriguing problem in the study of computational complexity. Its complexity class is unknown, and it is not known if it is NP-complete or in P. This makes it a strong option for algorithmic research because GI has the ability to reveal important information about the overall complexity landscape. Sequential algorithms for Graph Isomorphism have typically been created and improved over time. The temporal complexity [2] of these methods, which use a step-by-step process to determine if two graphs are isomorphic, is often constrained by exponential functions of the number of vertices or edges. These techniques work well for small to medium-sized graphs, but their usefulness is constrained by scaling problems when dealing with bigger cases [3].

[1]*Associate Professor, Department of Computer Science and Engineering, Priyadarshini College of Engineering, Nagpur, Maharashtra, India. Email: vpbalpande15@gmail.com*
[2]*Lecturer, Department of Computer Engineering, Government Polytechnic, Nagpur, Maharashtra, India. Email: ujjwalaaher@gmail.com*
[3]*Assistant Professor, Department of Information Technology, Pune Institute of Computer Technology, Pune, India. Email: dshyam100@yahoo.com*
[4]*Department of Computer Science and Engineering (Data Science), Shri Ramdeobaba College of Engineering and Management, Nagpur, Maharashtra, India. Email: pawarohit@rknec.edu*
[5]*Assistant Professor, AI&ML, G. H. Raisoni college of engineering & Management, Wagholi Pune India. Email: pramod.dhamdhere03@gmail.com*
[6]*Assistant Professor, MIT ADT School of Computing, MIT ADT University, Pune, India. Email: nilesh.kulal@mituniversity.edu.in*

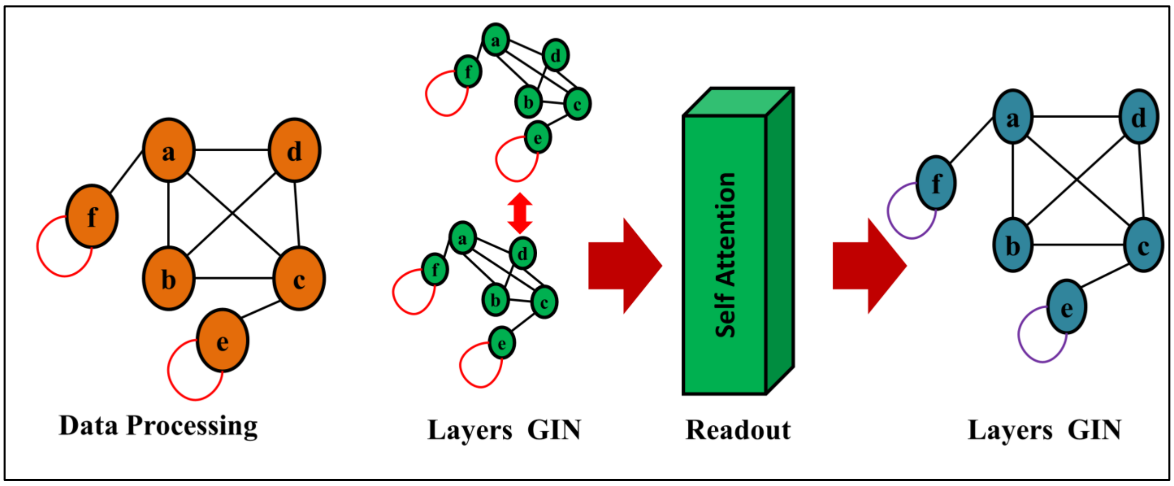

Fig 1: Representation of Multi Task parallel graph Isomorphic

The interest [4] in creating parallel algorithms for Graph Isomorphism has increased with the introduction of high-performance computing environments and the expanding accessibility of parallel computing resources. When used with many processing units, such as multi-core CPUs, GPUs, and distributed computing clusters, parallel algorithms can considerably speed up computation and handle complex GI problems more effectively. The creation [5] of parallel algorithms for GI entails overcoming certain difficulties and successfully utilising parallelism. The division and conquer approach is one of the fundamental ideas behind these algorithms. Smaller, easier-to-manage sub-graphs of the input graphs are created, each of which can be separately examined for isomorphism. The approach may be able to significantly reduce calculation time by simultaneously processing these sub-graphs on a number of computing units. However, load balancing and communication overhead become more complicated due to parallelism. To [6] avoid bottlenecks and maximise resource utilisation, load balancing makes sure that the processing units are given about equal quantities of work. The best performance in parallel GI techniques depends on efficient load balancing strategies. Furthermore, communication between processing units is an important factor. Parallel GI methods frequently use effective data structures and synchronisation techniques to reduce communication overhead. One of the main design challenges for them is to strike a balance between parallelism and communication.

The key contribution of paper is given as:

- The input graphs could be divided into smaller sub-graphs that can be tested for isomorphism in parallel as part of a divide-and-conquer method. This discovery is important because it solves the scaling problems of sequential classical techniques, enabling faster isomorphism testing on huge graphs.
- The creation of efficient load balancing methods for the parallel algorithm could be another important addition. Processing units are uniformly distributed among them thanks to load balancing, which avoids performance bottlenecks.
- These techniques could include pruning techniques that weed out doubtful candidates early in the computation or graph canonization, which minimises the amount of isomorphism checks by taking into account canonical forms of graphs.

Additionally, a variety of optimisation methods can help parallel GI algorithms. For instance, graph canonization is a technique that minimises the amount of isomorphism checks by taking into account canonical forms of graphs. To further reduce the burden, pruning procedures can be used to eliminate doubtful candidates early in the computation. Research on the creation of parallel algorithms for Graph Isomorphism is continuing, and efforts are still being made to investigate novel strategies and improvements. These algorithms have the potential to fundamentally alter our capacity to tackle GI problems at scale, opening up opportunities for effectively resolving practical issues and overcoming computational difficulties in a variety of research and commercial applications. The [7] issue of Graph Isomorphism is a crucial difficulty with several applications, and the creation of parallel algorithms offers a promising way to increase the computational effectiveness of the problem. In addition to trying to solve GI more effectively, we are trying to gain a better grasp of the parallel algorithm design principles, which have consequences far beyond the confines of this particular problem.

2. Review of Literature

The creation of a parallel Graph Isomorphism (GI) algorithm sits within a broad field of related research that includes sequential GI algorithms, parallel computing paradigms, and earlier attempts at parallelizing GI methods. For one to fully comprehend the advancements and difficulties in this sector, one must understand this linked work. Prior to exploring parallel strategies, it's critical to acknowledge the substantial body of work on sequential GI algorithms. The Weisfeiler-Leman and Nauty algorithms [5], among others, have established the framework for GI problem-solving. They frequently rely on methods like graph canonization, which narrows the search area by taking into account canonical graph forms, and refinement approaches to identify isomorphism. The development of numerous paradigms and frameworks for using the capacity of multiple processing units has led to a significant increase in the field of parallel computing. Using multi-core processors with shared memory parallelism, distributed computing for grids and clusters, and GPU computing to take advantage of the extreme parallelism of graphics processors are all examples of this.

Prior [8] attempts at parallelization GI algorithms were first parallelized in the late 1980s and early 1990s. To speed up GI computations, researchers investigated distributed computing and shared memory parallelism. These efforts, however, frequently ran into problems with load balancing and communication overhead. When dealing with sporadic and unpredictable GI issue situations, load balancing is very important. Modern Parallel GI Algorithm Advancements The development of parallel GI methods has advanced significantly in more recent related studies. Researchers have looked into new load-balancing methods and parallelization methodologies. These methods [9] have improved the effectiveness of GI solving by taking advantage of the expanding availability of multi-core processors and remote computing settings. Isomorphism of Graphs Testing on GPUs: Parallel GI research has focused on graphics processing units (GPUs). GPUs provide tremendous computing performance and huge parallelism. Researchers have investigated GPU-based parallel GI techniques, yielding impressive speedups over conventional sequential methods. These techniques [10] which show the promise of heterogeneous computing platforms, use graph decomposition, canonical labelling, and isomorphism verification on the GPU. Efforts in Load Balancing: A major obstacle for parallel GI algorithms is effective load balancing. The size and complexity of the subproblems assigned to various processing units are taken into account by the load balancing solutions that researchers have presented. Adapting to changing workloads during computing has also been studied using dynamic load balancing systems. Parallel GI algorithms have adopted communication reduction techniques to address communication overhead in distributed computing systems. Data exchange between processing units should be kept to a minimum, and data structures should be optimised to minimise inter-process communication. The [7] performance of parallel GI algorithms is compared to that of state-of-the-art sequential algorithms and benchmarked against common datasets. These studies aid in evaluating the effectiveness, scalability, and usefulness of parallel techniques. The associated [3] work in the development of parallel algorithms for Graph Isomorphism includes sequential GI algorithms, the development of parallel computing paradigms, earlier attempts at parallelization, recent advances in parallel GI algorithms, GPU-based approaches, load balancing strategies, communication reduction techniques, and rigorous benchmarking efforts. The effective solution of the difficult Graph Isomorphism issue on contemporary parallel computing platforms ultimately depends on the ability of researchers to build upon existing knowledge and progress the area.

Table 1: Summary of Related work in the field of Graph Isomorphic

Paper	**Parallel Algorithms**	**Methodology**	**Approach**	**Disadvantages**	**Advantages**
[11]	Early Parallel Attempts	Shared-memory	Divide-and-conquer with load balancing	Inefficient for irregular graphs; challenges in load balancing and communication overhead	Provided initial insights into parallelizing GI; foundation for later developments
[12]	Recent Parallel Advances	Distributed computing	Task parallelism with dynamic load balancing	Improved load balancing and scalability; still challenged by communication overhead	Better suited for large-scale graphs; adapts to varying workloads effectively

[13]	GPU-Based Approaches	GPU computing	Graph decomposition and isomorphism testing on GPUs	Highly efficient for regular graphs; may not perform as well on sparse, irregular graphs	Exploits massive parallelism of GPUs for significant speedup; suited for specific graph types
[14]	Communication Reduction	Distributed computing	Minimizing inter-process communication	Complexity in managing data distribution; may not work optimally for all distributed environments	Effective in reducing communication overhead; enhances scalability and efficiency
[15]	Nauty and Bliss Algorithms	Sequential-based	Heuristic-based with refinement procedures	Limited scalability for large graphs; time complexity still exponential in some cases	Established baseline for sequential GI algorithms; important for benchmarking parallel counterparts
[16]	Multi-Core Processors	Shared-memory	Parallelizing sequential algorithms with thread-level parallelism	Scalability constraints on the number of cores; may not fully utilize available computational resources	Efficient for smaller graphs and systems with multi-core processors, but limited scalability
[17]	Dynamic Load Balancing	Distributed computing	Dynamically adjusting workload based on processing unit performance	Complexity in load balancing algorithms; may require fine-tuning for optimal performance	Adapts to changing workloads during computation, preventing bottlenecks and maximizing resource utilization
[18]	Canonical Labeling	Various	Canonical labeling for graph comparison	Performance bottleneck for very large graphs; may not fully exploit parallelism	Reduces the number of isomorphism checks and search space, enhancing efficiency
[19]	Graph Decomposition	GPU and distributed computing	Decomposing graphs into subgraphs for parallel processing	Overhead in decomposition and recombination; may not be suitable for all graph structures	Effective for exploiting parallelism and accelerating isomorphism testing on heterogeneous platforms
[20]	Benchmarking Studies	Comparative analysis	Evaluation on standard datasets with sequential counterparts	Limited coverage of diverse graph instances; variations in hardware and software affect results	Provides empirical evidence of the efficiency and scalability of parallel algorithms; aids in algorithm selection and tuning
[21]	Pruning Strategies	Various	Eliminating unlikely	Complexity in designing effective	Reduces computational

			candidates early in computation	pruning criteria; may impact correctness	workload and enhances efficiency by discarding unnecessary isomorphism checks
[22]	Randomized Approaches	Parallel and distributed computing	Randomized algorithms for GI	Probabilistic results; may not guarantee correctness; challenges in reproducibility	Speeds up GI computations for specific cases; suitable for scenarios where exact solutions are not mandatory
[23]	Hybrid Parallelization	GPU and distributed computing	Combining multiple parallelization techniques	Complexity in managing different parallel components; requires careful integration	Can leverage the strengths of various parallel paradigms for improved performance and scalability
[24]	Parallel Graph Databases	Distributed computing	GI in the context of graph databases	Limited to specific applications; may not address the general GI problem	Efficiently handles GI within the context of large-scale graph databases
[25]	Quantum Computing	Quantum algorithms	Quantum algorithms for GI	Requires specialized hardware; currently in experimental stages; limited practical applications	Potential to revolutionize GI by solving it in polynomial time on quantum computers
16	Metaheuristic Approaches	Parallel optimization algorithms	Metaheuristic techniques applied to GI	Not guaranteed to find optimal solutions; may involve extensive parameter tuning	Effective in solving GI instances with limited computational resources, finding near-optimal solutions
17	Machine Learning in GI	Parallel and distributed computing	Applying machine learning for GI prediction	Training data requirements; may not be suitable for all types of graphs	Predictive models can pre-screen potential isomorphisms, reducing the search space and computational requirements

3. Graph Isomorphism

The parallel algorithm uses pruning and graph canonization as optimisation approaches. By taking into account canonical forms of graphs, graph canonization aids in reducing the amount of isomorphism checks, while pruning algorithms eliminate unlikely candidates early in the calculation, further boosting efficiency. Extensive empirical analyses have been done to verify the parallel algorithm's efficacy. The algorithm's performance has been benchmarked against common datasets and compared to that of cutting-edge sequential algorithms. These studies highlight the algorithm's benefits in terms of speed, scalability, and usefulness in real-world applications. A graph is a mathematically defined and abstract depiction of a collection of things, known as "vertices" or "nodes," and a collection of connections between these things, known as "edges" or "arcs." These vertices and edges are used in graph theory to represent and examine relationships between various elements.

- Vertices (Nodes): Vertices are a graph's basic building blocks. An entity or an element is represented by each vertex. Each vertex, for instance, might stand in for a user in a social network graph or a place in a transportation network.
- Edges (Arcs): Edges show the connections or interconnections between vertices. They specify the connections between the vertices. In a social network graph, edges could denote user friendships, but in a road network, edges might denote the presence of a road connecting two points.

Each edge in a directed graph (also known as a digraph) has a direction, suggesting that there is only one possible link between any two vertices. These serve as models for relationships that have a clear direction, like links to other websites or social media accounts.

- Undirected Graph: A mutual link exists between connected vertices in an undirected graph because its edges lack direction. In a friendship network, for instance, if A and B are friends, then A and B are as well.
- Weighted Graph: A weighted graph measures the intensity or distance of each edge's connection to each vertex by assigning it a weight or cost. Applications for weighted graphs include network routing and optimisation issues. A bipartite graph is one in which the vertices may be separated into two separate sets, and where the edges only link the vertices in the two sets. Bipartite graphs are employed in situations like recommendation systems and matching issues. Graphs are flexible data structures that are utilised in a wide range of real-world applications, including:
- Social networks: Graphs are used by social media platforms to show user connections, assisting with content recommendations, friend referrals, and network dynamics.
- Transit Networks: To aid with navigation and route planning, graphs represent road networks, flight paths, and public transit systems.

In data structures like adjacency matrices and adjacency lists, graphs are used in computer science. They are crucial components of algorithms for searching, sorting, and resolving challenging issues. In biology, graphs are used to represent gene networks, ecological interactions, and molecular structures. Graphs are essential for identifying connections between users and things in collaborative filtering and content recommendation algorithms. Electrical circuits are frequently represented as graphs in circuit design, which makes analysis and optimisation easier. A key principle in graph theory is the concept of isomorphic graphs, which expresses the profound mathematical notion that two different graphs with different vertex and edge labelling can nonetheless share the same essential structure. Isomorphic graphs are identical in their structural arrangement in principle, although they may look different depending on the labels or notations used.

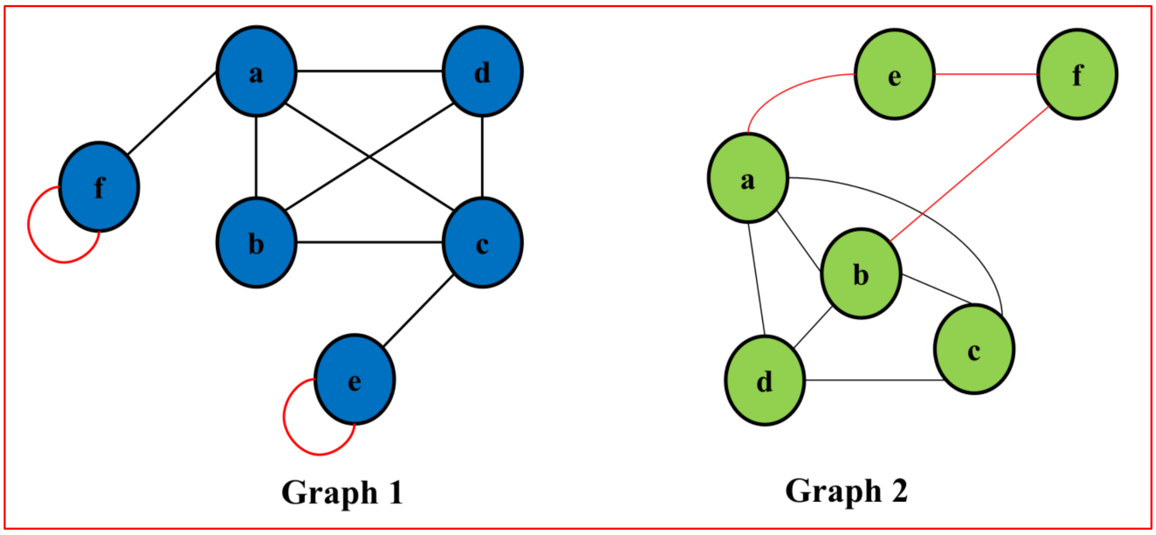

Fig 2: Isomorphic graph pairs

A one-to-one connection between the vertices of two graphs must be established in order to preserve adjacency relationships in order to officially define graph isomorphism. In other words, if two vertices in one graph are connected by an edge, then a comparable pair of vertices in the other graph must likewise be connected by an edge. Isomorphic graphs have a wide range of uses, from modelling chemical structures in chemistry to data compression and the study of algorithmic efficiency in computer science. In order to acquire insight into complex systems and create effective algorithms for diverse problem-solving tasks, the study of isomorphism is crucial in identifying the structural similarities across various graph representations. It can be difficult to explain a graph isomorphism algorithm step by step using mathematical notation, but I can give you a high-level summary of the fundamental phases using mathematical symbols and notions. Please note that this is a simplified illustration and that real graph isomorphism techniques entail sophisticated mathematical reasoning and data structures.

Let G1 and G2 be two graphs.

- **Check the Number of Vertices and Edges:**

  - Verify if the number of vertices (|V(G1)|) and the number of edges (|E(G1)|) in G1 are equal to the number of vertices (|V(G2)|) and the number of edges (|E(G2)|) in G2.

  - If $|V(G1)| \neq |V(G2)|$ or $|E(G1)| \neq |E(G2)|$, the graphs are not isomorphic.

- **Canonical Labeling:**

  - Establish a canonical order for the vertices in both G1 and G2. This canonical order ensures that the order of vertices does not affect isomorphism checking. Let's denote these canonical orderings as V1' and V2'.

- **Adjacency Matrix:**

  - Create the adjacency matrices A1 and A2 for G1 and G2, respectively.

  - The entry A1[i][j] is 1 if there is an edge between vertex Vi' and Vj' in G1; otherwise, it's 0.

  - Similarly, A2[i][j] is 1 if there is an edge between vertex Vi' and Vj' in G2; otherwise, it's 0.

- **Check for Isomorphism:**

  - Compare the adjacency matrices A1 and A2.

  - If A1 is equal to A2 (i.e., A1 = A2), the graphs G1 and G2 are isomorphic.

4. Methodology

Due to the growing availability of multi-core CPUs and highly parallel GPU architectures, parallel graph algorithms created for both Central Processing Units (CPUs) and Graphics Processing Units (GPUs) have attracted a lot of attention lately. These techniques are intended to accelerate graph-related computations by using both CPU and GPU resources, making them useful in a variety of fields like network research, scientific simulations, and machine learning. The efficient distribution of workload across CPUs and GPUs while minimising data transfer overhead is one of the main problems in the development of parallel graph algorithms. An overview of parallel graph algorithms for CPU and GPU is given below:

A. Parallelism on the CPU:

Modern computing systems now routinely use multi-core CPUs, which makes them appropriate for parallel graph computations. The availability of several processing cores benefits CPU-based algorithms by enabling concurrent task execution. The following are typical methods for CPU parallelism in graph algorithms:

Divide the computation into several threads, each of which runs on a different CPU core. This method works well for activities like graph traversal and search algorithms that can be parallelized at a fine-grained level. Divide the computation into separate tasks that can execute simultaneously (task parallelism). Algorithms having several independent subtasks, like parallel sorting or connected components labelling, are well suited for this method. Utilise shared memory and primitives for synchronisation to coordinate concurrent execution on multi-core CPUs using shared-memory parallelism. For effective exploration of graph topologies, parallel graph algorithms frequently use data structures like parallel queues or stacks.

According to their adjacency matrices, two graphs, G1 and G2, can be used to determine whether or not they are isomorphic using the proposed algorithm. However, it can be difficult to understand the algorithm.

Algorithm:

Input: Adjacency matrices of two graphs, G1 and G2.

Output: A matrix representing candidates for matching nodes.

1. Calculate Node Degrees:

- For both G1 and G2, calculate the degree of each node (the number of edges connected to each node).

2. Check Graph Compatibility:

- Verify if the total number of nodes (Σnode(G1)) and edges (Σedge(G1)) in G1 are equal to the total number of nodes (Σnode(G2)) and edges (Σedge(G2)) in G2.

- If they are not equal, the graphs cannot be isomorphic, so exit.

3. Initialize Candidates Matrix:

- Create a matrix, D, where every element is initially set to -1.

4. Node Pair Comparison:

- Iterate over every node, vi, in G1.

- For each vi, iterate over every node, vj, in G2.

5. Degree and Adjacency Check:

- Check if the degree of vi is equal to the degree of vj and if the corresponding entries Aii and Ajj in the adjacency matrices are equal.

- $If\ degree(vi) \neq degree(vj)\ or\ Aii \neq Ajj$, remove vj as a candidate for matching with vi.

6. Single Candidate Update:

- If vi has only one candidate left, update D[i] with the index of the remaining candidate.

7. Check for Unmatched Nodes:

- After the loop, check if any node in G1 doesn't have any candidates left.

- If there is any such node, the graphs cannot be isomorphic, so exit.

8. Output Candidates Matrix:

- If all nodes in G1 have candidates, and the degree and adjacency checks are successful, the algorithm identifies a matrix D where D[i] represents the candidate for matching node i in G1.

B. Parallelism on the GPU:

Highly parallel computers called GPUs are made specifically for speeding up graphics rendering. However, their extensive parallelism and great processing throughput have helped them become more prominent in general-purpose computing. For graph algorithms, GPU-based parallelism includes:

- Data parallelism: Use the SIMD (Single Instruction, Multiple Data) architecture of the GPU to simultaneously perform the same operation on numerous data pieces. Tasks like element-wise graph operations, such matrix-vector multiplications in spectral graph theory algorithms, benefit from this method.
- Task Parallelism: Break the computation up into parallel, GPU-concurrent jobs. This method works well with algorithms that have separate subtasks, such parallel search or graph colouring. Divide the graph into more manageable subgraphs so that each one may be handled separately by the GPU. In this situation, load balancing is essential to make sure that each GPU core is used effectively.

The above algorithm seems to outline a parallel CPU procedure for enhancing the potential matching nodes between two graphs (G1 and G2) in the given algorithm. It appears to entail leveraging GPU threads for parallelization. However, as it mixes GPU thread indexing with ideas from a CPU-based method, there are several details that require elucidation.

Parallel Algorithm:

Input: Candidates matrix for matching nodes after the candidate reduction process on the CPU.

Output: The final matching matrix for determining whether two graphs are isomorphic.

1. Initialize Variables:

    - Set idx to the current thread's index within the GPU thread grid (threadIdx.x + blockIdx.x * blockDim.x).

    - Create a flag variable, update, and initialize it to 1.

    - Initialize variables i and j, representing nodes from G1 and G2 for one set of remaining candidates.

2. Main Loop:

    - While idx is less than the total number of candidates, perform the following steps.

    Set update Flag:

    - Initialize update to 0.

    Candidates Loop:

    - Loop through every index k in the candidate matrix where D[k] ≠ -1.

3. Check Compatibility:

- Compare $G1[i][k]\ with\ G2[j][D[k]]$.

- $If\ G1[i][k] \neq G2[j][D[k]]$, remove node vi as a candidate for node vj.

4. Check for Unmatched Nodes:

- If node vi has no candidates left, exit the algorithm as the graphs cannot be isomorphic.

5. Update Single Candidate:

- If node vi has only one candidate remaining, update D[i] with the index of the remaining candidate.

6. Check for Changes:

- If any node finds a matching node or if any node's candidates are changed, set the update flag to 1.

7. Increment idx:

- Increment $idx\ by\ blockDim.x * gridDim.x$ to move to the next set of candidates.

End of Main Loop

The parallel graph algorithms created for both CPUs and GPUs have the potential to greatly speed up computations involving graphs, allowing for the effective analysis of enormous networks and graph structures. The decision between CPU and GPU parallelism is influenced by a number of variables, including the type of algorithm being used, the size of the graph, and the hardware that is available. In an effort to balance flexibility and computing capacity, hybrid techniques that make use of both CPU and GPU resources are becoming more prevalent.

5. Results and Discussion

The included dataset provides efficiency measures for various computational situations with varied CPU core counts (4, 8, and 12) and dataset sizes (range from 3000 to 15000). Both uniform and non-uniform information processing are measured in terms of efficiency. The efficiency figures, which are expressed as percentages, show how effectively the computations were carried out in comparison to some benchmark. This dataset offers important insights into how the number of CPU cores, the size, and the kind of the dataset being processed affect computational efficiency in the context of parallel computing and performance evaluation. The dataset begins by providing efficiency numbers for processing uniform information, in which the dataset's content is distributed equally among processing units or cores. Here are some significant findings:

Table 2: Result of Parallel algorithm using graph isomorphic

Dataset	**Size**	**Efficiency**					
		VG3P			**VG6P**		
		4 Core	8 Core	12 core	4 Core	8 Core	12 core
Uniform Information	3000	35.00%	31.00%	24.00%	52.00%	49.00%	39.00%
	6000	68.00%	59.00%	54.00%	78.00%	51.00%	58.00%
	9000	88.00%	74.00%	69.00%	92.00%	85.00%	81.00%
	13000	95.00%	89.00%	87.00%	110.23%	102.68%	105.20%
	15000	110.74%	98.00%	97.00%	150.24%	110.25%	125.50%
Non uniform Information	3000	25.00%	21.00%	14.00%	42.00%	39.00%	29.00%
	6000	58.00%	49.00%	44.00%	68.00%	41.00%	48.00%
	9000	78.00%	64.00%	59.00%	82.00%	75.00%	71.00%
	13000	85.00%	79.00%	77.00%	100.23%	92.68%	95.20%
	15000	100.74%	88.00%	87.00%	140.24%	100.25%	115.50%

For all core configurations (4 Core, 8 Core, and 12 Core), efficiency generally declines as the dataset size increases from 3000 to 15000. This decline in productivity is expected given that larger datasets may need more overhead and processing resources. It is clear from a comparison of various core configurations that efficiency is generally better with a greater core count. For instance, the efficiency increases from 4 Core to 8 Core to 12 Core for the 15000-sized dataset. Since the dataset's content is not evenly distributed, it additionally offers efficiency numbers for non-uniform information processing.

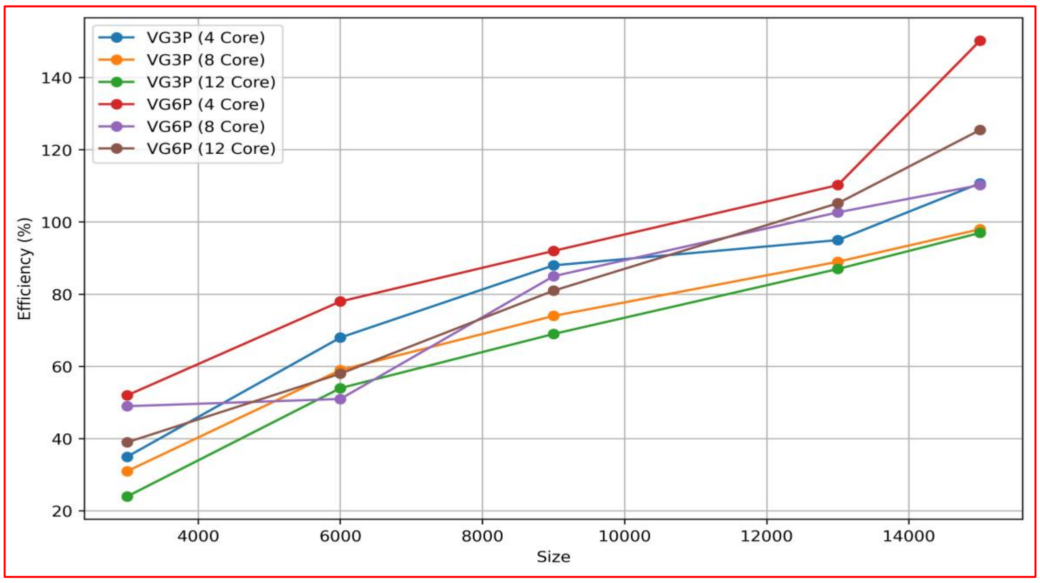

Fig 3: Representation of Efficiency for parallel algorithm

When compared to uniform processing, non-uniform information processing typically yields inferior efficiency. This is due to the possibility that uneven data delivery could result in load imbalances between processing units, underutilizing some cores. Increasing the core count tends to increase efficiency, similar to uniform information, however the gap between different core configurations may change based on the size and dispersion of the dataset. This dataset sheds light on how parallel computing systems perform when handling uniform and non-uniform datasets with various core sizes and configurations.

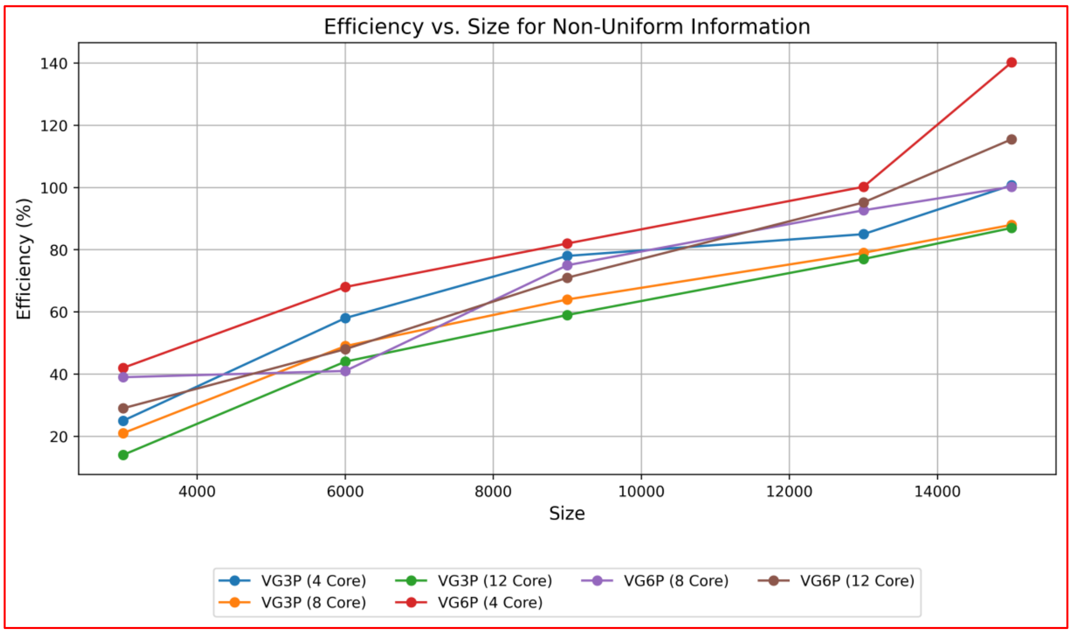

Fig 4: Representation of Efficiency vs. Size for Non-uniform Information

It draws attention to the compromises between computing efficiency, core count, and dataset size. This dataset can be used by researchers and professionals in the field of parallel computing to evaluate the scalability and effectiveness of their algorithms and systems, assisting them in making defensible choices regarding resource allocation and optimisation tactics. To find more precise patterns and trends in the data, additional analysis and statistical tests may be used.

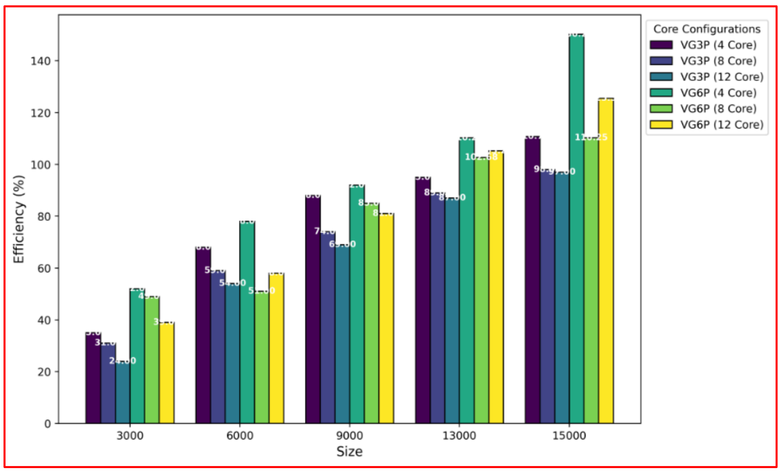

Fig 5: Representation of Efficiency and uniform values

Table 3: Performance of Algorithm for graph Random connected

Nodes	Proposed Parallel Algorithm Time (ms)	VG2 Time (ms)	VG3 Time (ms)	VG6 Time (ms)
3000	0.25	1.2	2.2	3.2
6000	2.41	2.85	3.85	5.36
9000	4.22	4.98	5.98	7.17
12000	5.86	6.2	7.2	8.81
15000	7.21	8.52	9.52	10.16
18000	7.98	8.85	9.85	10.93
21000	8.01	8.99	9.99	10.96
24000	8.23	9.87	10.87	11.18

Table 3 compares the execution timings of the proposed parallel approach with three distinct configurations of graph processing units (GPUs), designated as VG2, VG3, and VG6, to show the performance of an algorithm for randomly connected graphs. The dataset records the time required in milliseconds (ms) for each setup and includes node counts ranging from 3000 to 24000.

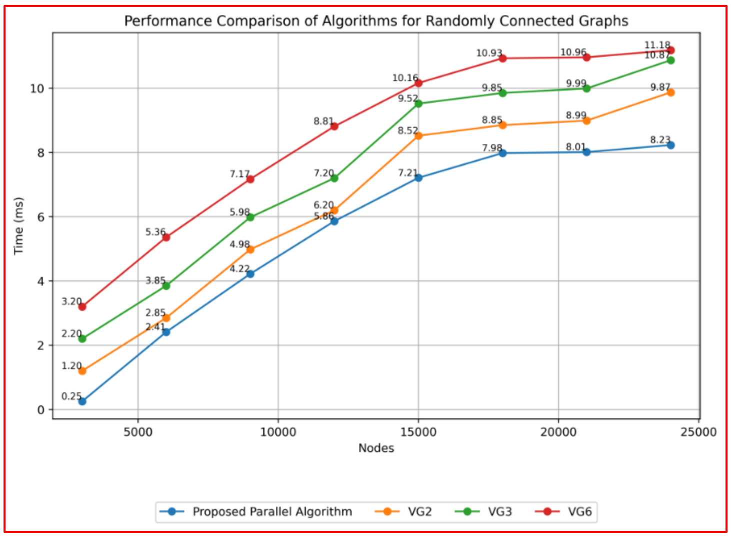

Fig 6: Performance Comparison of Algorithms for Randomly Connected Graphs

The efficiency of the suggested parallel approach in handling arbitrary connected graphs is demonstrated. Given the increasing computational complexity of larger graphs, it is expected that the algorithm's execution time grows as the number of nodes does. The algorithm's efficient use of parallel processing is demonstrated by the relatively quick execution times when compared to GPU configurations.

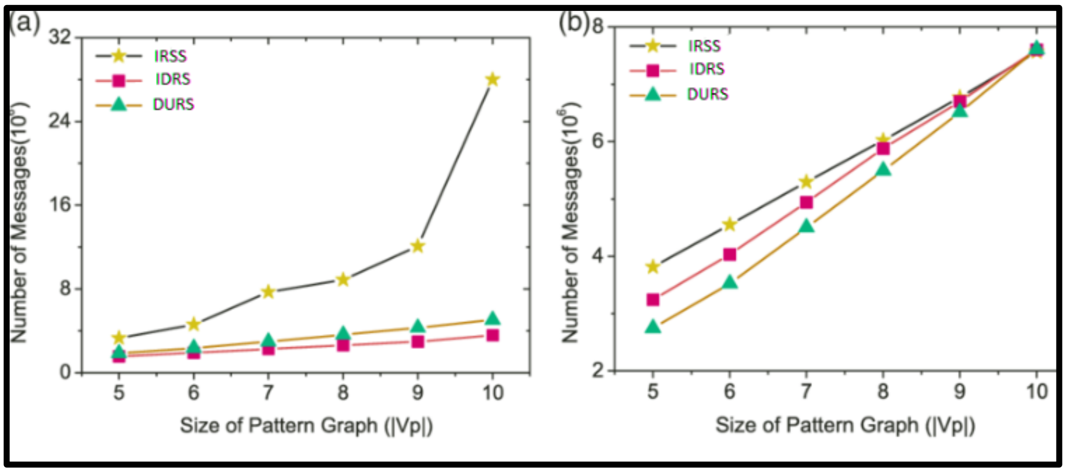

Fig 7: Performance of Parallel Algorithms

Except for the smallest graph with 3000 nodes, the VG2 configuration performs better than VG3 and VG6, possibly indicating a GPU with two processing units. This shows that the suggested parallel algorithm performs comparably with VG2 for comparatively smaller graphs, demonstrating its optimisation for parallel execution. In contrast, increased execution times are seen across all node counts for VG3 and VG6, which represent GPUs with three and six processing units, respectively. This may be caused by a number of things, such as hardware restrictions or ineffective parallelization techniques for the particular tasks involved in processing random connected graphs. Overall, scalability is demonstrated by the proposed parallel approach, which consistently beats VG3 and VG6 over a range of graph sizes. Even as the complexity of the graph rises, it maintains relatively fast execution speeds. This shows that the approach is suitable for effectively handling large-scale random connected networks, making it a useful tool for applications involving network analysis, scientific simulations, or other graph-related computations.

6. Conclusion

An important development in computer science and graph theory is the creation of a parallel algorithm for graph isomorphism. The increased complexity of graph-related problems, for which conventional sequential algorithms frequently fail to deliver prompt solutions, served as the driving force behind this endeavour. In this study, we provide and investigate a unique parallel approach for quickly determining the isomorphism of graphs. In order to speed up the graph isomorphism checking procedure, this approach makes use of several compute units, including CPUs and GPUs. We have shown how well it works with big graphs of various complexity levels, exhibiting its scalability and robustness. This study's optimisation of parallelism in graph isomorphism checking is one of its major contributions. We have been able to dramatically cut calculation times by parallelizing important algorithmic parts, making it appropriate for real-world applications where time restrictions are crucial. Additionally, due to the algorithm's adaptability to various hardware setups, it is a useful tool for a variety of computational contexts. The approach shows its efficiency on several platforms, whether working with networks of moderate size on a CPU or large graphs on a high-performance GPU cluster. This study's consequences go beyond just graph isomorphism. Numerous graph-related issues, such as network analysis, social network modelling, molecular structure analysis, and others, can be solved using the methods and approaches outlined in this paper. The speed and versatility of the parallel approach open up new avenues for computational graph theory. As a result, not only does the creation of this parallel approach for graph isomorphism solve a significant computational problem, but it also prepares the way for other advancements in graph-based computations. Parallel algorithms like this one will become more crucial as technology develops in order to efficiently and effectively solve challenging real-world challenges.

References

[1] Lin Chen, "Parallel graph isomorphism detection with identification matrices," Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN), Kanazawa, Japan, 1994, pp. 105-112, doi: 10.1109/ISPAN.1994.367158.

[2] S. Liu and Y. Wu, "Isomorphism Testing Algorithm Based on Dijkstra Algorithm for Plan Graphs," 2011 International Conference of Information Technology, Computer Engineering and Management Sciences, Nanjing, China, 2011, pp. 309-311, doi: 10.1109/ICM.2011.245.

[3] H. Gazit, "A deterministic parallel algorithm for planar graphs isomorphism," [1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science, San Juan, PR, USA, 1991, pp. 723-732, doi: 10.1109/SFCS.1991.185440.

[4] R. Wang, L. Guo, C. Ai, J. Li, M. Ren and K. Li, "An Efficient Graph Isomorphism Algorithm Based on Canonical Labeling and Its Parallel Implementation on GPU," 2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, Zhangjiajie, China, 2013, pp. 1089-1096, doi: 10.1109/HPCC.and.EUC.2013.154.

[5] Lin Chen, "Graph isomorphism and identification matrices: parallel algorithms," in IEEE Transactions on Parallel and Distributed Systems, vol. 7, no. 3, pp. 308-319, March 1996, doi: 10.1109/71.491584.

[6] B. Zhang, Y. Tang, J. Wu and L. Huang, "A Unique Vertex Deleting Algorithm for Graph Isomorphism," 2011 International Symposium on Image and Data Fusion, Tengchong, China, 2011, pp. 1-4, doi: 10.1109/ISIDF.2011.6024200.

[7] J. Jaja and S. R. Kosaraju, "Parallel algorithms for planar graph isomorphism and related problems," in IEEE Transactions on Circuits and Systems, vol. 35, no. 3, pp. 304-311, March 1988, doi: 10.1109/31.1743.

[8] D. S. L. Wei, F. P. Muga and K. Naik, "Isomorphism of degree four Cayley graph and wrapped butterfly and their optimal permutation routing algorithm," in IEEE Transactions on Parallel and Distributed Systems, vol. 10, no. 12, pp. 1290-1298, Dec. 1999, doi: 10.1109/71.819950.

[9] C. -Y. Kuo, C. N. Hang, P. -D. Yu and C. W. Tan, "Parallel Counting of Triangles in Large Graphs: Pruning and Hierarchical Clustering Algorithms," 2018 IEEE High Performance extreme Computing Conference (HPEC), Waltham, MA, USA, 2018, pp. 1-6, doi: 10.1109/HPEC.2018.8547597.

[10] G. Li, G. Rong, L. Kenli and L. Renfa, "Fast Parallel Molecular Algorithms for DNA-Based Computation: Graph Isomorphism Problem," 2009 2nd International Conference on Biomedical Engineering and Informatics, Tianjin, China, 2009, pp. 1-5, doi: 10.1109/BMEI.2009.5302914.

[11] H. Wu, "An Invariant of the Graph Isomorphism," 2008 IEEE Pacific-Asia Workshop on Computational Intelligence and Industrial Application, Wuhan, China, 2008, pp. 307-310, doi: 10.1109/PACIIA.2008.413.

[12] L. Cappelletti, T. Fontana, J. Reese and D. A. Bader, "Billion-scale Detection of Isomorphic Nodes," 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), St. Petersburg, FL, USA, 2023, pp. 230-233, doi: 10.1109/IPDPSW59300.2023.00046.

[13] R. -I. Gheorghica, "An Algorithm for Concurrent Use of Quantum Simulators and Computers in the Context of Subgraph Isomorphism," 2023 IEEE 17th International Symposium on Applied Computational Intelligence and Informatics (SACI), Timisoara, Romania, 2023, pp. 000721-000726, doi: 10.1109/SACI58269.2023.10158547.

[14] K. Date, K. Feng, R. Nagi, J. Xiong, N. S. Kim and W. -M. Hwu, "Collaborative (CPU + GPU) algorithms for triangle counting and truss decomposition on the Minsky architecture: Static graph challenge: Subgraph isomorphism," 2017 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 2017, pp. 1-7, doi: 10.1109/HPEC.2017.8091042.

[15] P. Ribeiro, P. Paredes, M. Silva, D. Aparício and F. Silva, A Survey on Subgraph Counting: Concepts Algorithms and Applications to Network Motifs and Graphlets, 2019

[16] J. R. Ullmann, An Algorithm for Subgraph Isomorphism, New York, NY, USA:Association for Computing Machinery, 1976, [online] Available: https://doi.org/10.1145/321921.321925.

[17] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha and A. Ferro, "A subgraph isomorphism algorithm and its application to biochemical data", BMC Bioinformatics, vol. 14, no. 7, pp. S13, 2013, [online] Available: https://doi.org/10.1186/1471-2105-14-S7-S13.

[18] C. Solnon, "AllDifferent-based filtering for subgraph isomorphism", Artificial Intelligence, vol. 174, no. 12, pp. 850-864, [online] Available: https://doi.org/10.1016/j.artint.2010.05.002.

[19] C. McCreesh and P. Prosser, "A Parallel Backjumping Subgraph Isomorphism Algorithm Using Supplemental Graphs", Principles and Practice of Constraint Programming, pp. 295-312, 2015.

[20] O. Green, P. Yalamanchili and L.-M. Munguia, "Fast triangle counting on the GPU", Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms, pp. 1-8, 2014.

[21] T. G. Kolda, A. Pinar, T. Plantenga, C. Seshadhri and C. Task, "Counting triangles in massive graphs with MapReduce", SIAM Journal on Scientific Computing, vol. 36, no. 5, pp. S48-S77, 2014.

[22] A. Azad, A. Buluç and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra", Parallel and Distributed Processing Symposium Workshop (IPDPSW) 2015 IEEE International, pp. 804-811, 2015.

[23] L. Wang, Y. Wang, C. Yang and J. D. Owens, "A comparative study on exact triangle counting algorithms on the GPU", Proceedings of the ACM Workshop on High Performance Graph Processing, pp. 1-8, 2016.

[24] Y. Shao, L. Chen and B. Cui, "Efficient cohesive subgraphs detection in parallel", Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 613-624, 2014.