IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 30 March 2024, accepted 18 April 2024, date of publication 25 April 2024, date of current version 2 May 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3393771

RESEARCH ARTICLE

Distributed Memory Implementation of Bron-Kerbosch Algorithm

TEJAS RAVINDRA ROTE, MURUGAN KRISHNAMOORTHY, ANSH BHATIA, RISHU YADAV, AND S. P. RAJA

School of Computer Science and Engineering (SCOPE), Vellore Institute of Technology, Vellore, Tamil Nadu 632014, India

Corresponding author: Murugan Krishnamoorthy (murugan.k@vit.ac.in)

This work was supported by the Vellore Institute of Technology.

ABSTRACT This paper proposes a parallel implementation of the Bron-Kerbosch algorithm, which finds all maximal cliques in large, complex graphs using CPU and thread-level parallelism and distributed memory with multiple cores. With the growing size and complexity of modern graphs, sequential algorithms can become impractical, making parallel computation a promising solution for reducing duration and enhancing scalability. Decomposing the graph into smaller components and distributing the workload across multiple threads using task parallelism is the proposed method. To optimise the algorithm's efficacy across processes, load-balancing techniques are also investigated. The modified Bron-Kerbosch algorithm is implemented and evaluated on a variety of large graphs, exhibiting significant runtime and scalability enhancements over the sequential version. The conclusion of the paper discusses the prospective applications of the parallel Bron-Kerbosch algorithm in a variety of domains, including social network analysis, bio informatics, and network security. Overall, this research contributes to the expanding body of work on parallel computing and graph analysis by emphasising the advantages of CPU and thread-level parallelism for efficiently solving complex computational problems.

INDEX TERMS Distributed memory, Bron-Kerbosch algorithm, parallel computing, high-performance computing, graph theory, maximal cliques, social network analysis, bio-informatics, network security, performance optimization, scalability, compute nodes, compute threads.

I. INTRODUCTION

The goal of this paper is to propose a parallel implementation of the Bron-Kerbosch algorithm for locating all maximal cliques in large graphs using CPU and parallel programming techniques such as task parallelism which involves decomposing the algorithm into smaller tasks to be executed concurrently and data parallelism by dividing the data across multiple processors to perform the same operation independently. This approach is effective for algorithms with many independent tasks or large data sets such as graphs.

Maximal clique is a subgraph where every vertex is connected to every other vertex in the subgraph and which cannot be extended by including one more adjacent vertex. Finding all maximal cliques in a graph is NP-hard because the number of potential cliques grows exponentially with the graph size, making it computationally intensive to identify every clique without a known polynomial-time solution.

Fig 1 and Fig 2 illustrate the NP-complete problem of locating maximal cliques in graphs. It emphasises the significance of locating maximum cliques and the computational complexity that makes sequential algorithms impracticable for large datasets.

The Fig 3 contains multiple cliques, but only the blue and red polygons are maximal cliques. By adding more neighbouring vertices to the set, maximal cliques cannot be created further. It is essential to observe that maximum cliques and maximal cliques are distinct. A maximum clique is a clique in a network that cannot contain more vertices than any other clique, which is referred to as the clique count of a graph. Although a maximum clique is always maximal, the converse is not always the case.

To circumvent this, the authors propose a modified parallel version of the Bron-Kerbosch algorithm to decrease the total

The associate editor coordinating the review of this manuscript and approving it for publication was Barbara Masini.

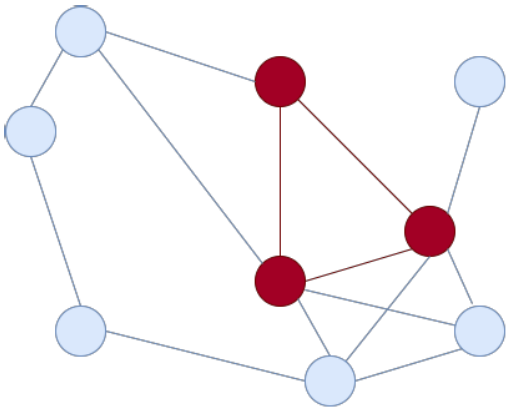


FIGURE 1. Example of maximal cliques.

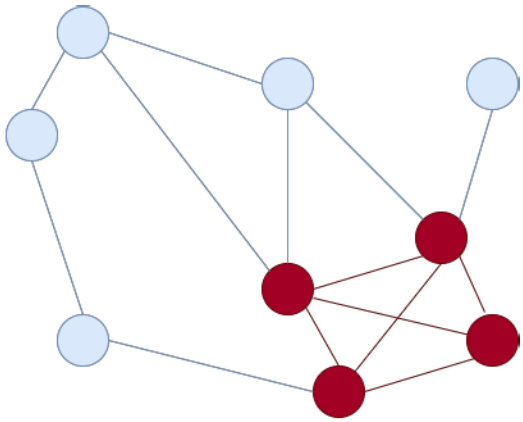


FIGURE 2. Example of maximum cliques.

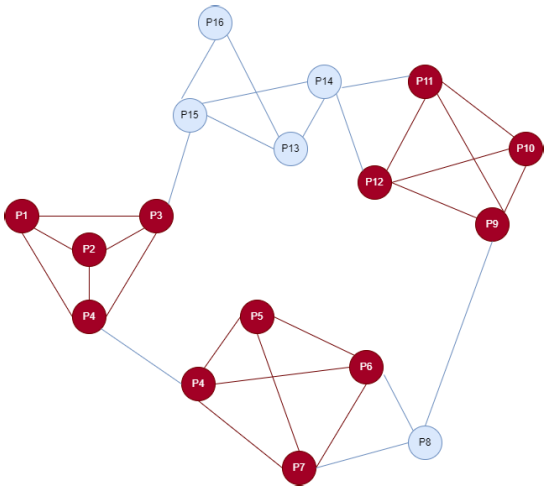


FIGURE 3. Graph having both maximum and maximal cliques.

duration for locating the largest clique in a graph using parallel computing.

The novelty of this research paper lies in its integration of CPU and thread-level parallelism to optimize the Bron-Kerbosch algorithm, contrasting previous works that predominantly focused on GPU-accelerated computations. While GPU acceleration has been shown to significantly reduce runtime by leveraging massive parallelism, it presents challenges including higher development complexity, greater demand for specialized hardware, and potential underutilization in environments not optimized for GPU resources.

Our methodology addresses these challenges by employing a more accessible, cost-effective approach that exploits the ubiquitous nature of multi-core CPUs and the inherent parallelism of modern computing environments.

Moreover, our work contributes to the field by enhancing load balancing and scalability through sophisticated thread management and data distribution techniques, ensuring efficient utilization of CPU resources. This is particularly relevant for applications in domains without access to specialized GPU hardware or where the economic implications of deploying GPU-based solutions are prohibitive.

Overall, the research paper presents a novel approach to parallelizing the Bron-Kerbosch algorithm that leverages the power of CPU and thread-level parallelism to accelerate the computation and improve load balancing, with potential applications in various domains, including social network analysis, bioinformatics, and network security. This paper presents a parallel implementation of the Bron-Kerbosch algorithm, which utilizes CPU and thread-level parallelism and distributed memory with multiple cores to find all the largest cliques in a large complex graph. The key contribution of our methodology, set against GPU-accelerated methods, includes addressing the accessibility and scalability challenges while maintaining competitive performance, marking a significant advancement in the parallel computing landscape for the Bron-Kerbosch algorithm.

II. RELATED WORKS

Wei et al. [1] addressed the problem of accelerating the Bron-Kerbosch algorithm for maximal clique enumeration using GPUs. The Bron-Kerbosch algorithm is a widely-used algorithm for finding all maximal cliques in an undirected graph, but it can be computationally expensive for large graphs. They proposed a parallel implementation of the algorithm using GPUs to significantly reduce the computation time. Their approach involved partitioning the input graph into sub-graphs and processing each sub-graph in parallel on a GPU, which allowed for efficient processing of large graphs with billions of edges. The authors evaluated their approach on various real-world datasets and showed that it achieved significant speedup compared to CPU-based implementations, demonstrating the effectiveness of their approach in accelerating the Bron-Kerbosch algorithm. Conte and Tomita [2] analyzed the overall and delay complexity of the CLIQUES and Bron-Kerbosch algorithms, which are two widely-used algorithms for finding all maximal cliques in an undirected graph. They presented a theoretical analysis of these algorithms and derived upper and lower bounds on their overall and delay complexity. Their analysis

showed that the Bron-Kerbosch algorithm has a higher overall complexity but a lower delay complexity than the CLIQUES algorithm. Moreover, the authors proposed a modification to the Bron-Kerbosch algorithm to reduce its overall complexity while maintaining its low delay complexity. Their approach involved pruning the search space by exploiting the structure of the input graph. The authors also evaluated their approach on various real-world datasets and showed that it achieved significant speedup compared to the original Bron-Kerbosch algorithm. The study provides valuable insights into the theoretical complexity of the CLIQUES and Bron-Kerbosch algorithms and offers a practical approach to improve the efficiency of the Bron-Kerbosch algorithm. The paper by Brosowsky et al. [3] explores the problem of finding maximal skew sets of lines on a Hermitian surface, which has applications in coding theory and algebraic geometry. They introduced a new algorithm based on the Bron-Kerbosch algorithm for solving this problem efficiently. Their approach involved modifying the Bron-Kerbosch algorithm to work on the incidence graph of lines and points on the Hermitian surface, and using this modified algorithm to find maximal skew sets of lines. The authors showed that their algorithm outperformed existing algorithms for the problem on various test cases. Their study contributes to the development of efficient algorithms for finding maximal skew sets of lines on a Hermitian surface, which has important applications in various fields. Zaitseva et al. [4] examines the problem of finding a compromise solution in economic competition, where multiple parties compete to achieve their own objectives while facing conflicting constraints. The authors proposed a new approach based on the compromise programming method to find a compromise solution that satisfies the objectives of all parties and minimizes the deviation from their ideal solutions. Their approach involved formulating the problem as a mathematical optimization problem and solving it using a specialized algorithm. The authors demonstrated the effectiveness of their approach on various test cases and showed that it outperformed existing approaches for finding a compromise solution in economic competition. Their study provides a valuable contribution to the field of economics and offers a practical approach to addressing the problem of finding a compromise solution in economic competition. The paper by Besta et al. [5] addresses the problem of efficiently performing graph mining on processing-in-memory systems. They proposed a new instruction set architecture (ISA) called Sisa, which is designed specifically for graph processing and is set-centric in nature. Their approach involved designing a set of new instructions that operate on sets of data, rather than individual data elements, and implementing them on a processing-in-memory system. The authors demonstrated the effectiveness of their approach on various test cases and showed that it outperformed existing graph processing architectures. Their study provides a valuable contribution to the field of computer architecture and offers a practical solution to the problem of efficient graph mining on processing-in-memory systems. The paper by Sade and Cohen [6] addresses the problem of diverse enumeration of maximal cliques, which involves finding all possible subsets of nodes in a graph that form complete subgraphs, while ensuring that the enumerated cliques are diverse in terms of their size and structure. They proposed a new algorithm based on a divide-and-conquer strategy that combines different enumeration techniques to achieve both efficiency and diversity. The authors demonstrated the effectiveness of their approach on various test cases and showed that it outperformed existing algorithms for diverse enumeration of maximal cliques. Their study provides a valuable contribution to the field of graph mining and offers a practical solution to the problem of diverse enumeration of maximal cliques. The paper by Coppola and Elgazzar [7] proposes novel machine learning algorithms for detecting centrality and cliques in YouTube social networks. They first extract the social network from YouTube data and then apply their algorithms to identify the most important nodes (i.e., those with high centrality) and cliques (i.e., groups of nodes with strong connections). The authors show that their approach outperforms traditional algorithms for centrality and clique detection in terms of accuracy and efficiency. The study demonstrates the potential of machine learning techniques for analyzing large-scale social networks and provides valuable insights into the structure and dynamics of YouTube communities. The paper by Prinz [8] presents a method for computing the maximum common edge sub-graph (MCES) of two molecular graphs. The MCES is an important problem in chemistry as it can help identify structural similarities between molecules, which is useful for drug design and other applications. The author proposes an efficient algorithm based on a dynamic programming approach that can handle large molecular graphs with thousands of vertices and edges. The proposed algorithm has been shown to outperform existing methods for MCES computation on a benchmark dataset. The study provides a valuable contribution to the field of computational chemistry and highlights the potential of dynamic programming techniques for solving complex graph problems. The paper by Pasternak et al. [9] proposes a method for identifying protein function and functional links based on large-scale co-occurrence patterns. The authors use data from the STRING database to construct a co-occurrence network of proteins, where edges between nodes represent the frequency of co-occurrence across different biological processes. They then apply a community detection algorithm to identify clusters of proteins with similar co-occurrence patterns and infer their functional annotations. The proposed method was shown to achieve high accuracy in predicting protein functions and identifying functional links between proteins. The paper by Gianinazzi et al. [10] presents parallel algorithms for finding large cliques in sparse graphs. The authors propose two algorithms based on the Bron-Kerbosch algorithm, a popular algorithm for clique enumeration. The first algorithm uses a vertex-centric approach and the second

algorithm uses a set-centric approach, where the set of candidate vertices for each recursive call is stored explicitly. The authors demonstrate that their parallel algorithms can achieve high speedup on large-scale sparse graphs, outperforming existing state-of-the-art algorithms for clique enumeration. The study highlights the importance of parallel computing in graph algorithms and provides useful insights for optimizing clique enumeration on modern parallel architectures. The paper by Almasri et al. [11] presents a parallelization approach for maximal clique enumeration on GPUs. The authors propose a new parallel algorithm that exploits the parallelism of GPUs and improves the performance of the Bron-Kerbosch algorithm. The approach is based on a workload balancing scheme that distributes the computation across multiple GPUs. The experimental results show that their approach outperforms existing CPU-based implementations and achieves significant speedup compared to serial GPU implementations. In the paper by Zhou et al. [12], the authors focus on the problem of enumerating maximal k-plexes, a generalization of the maximal clique problem. The paper presents a new algorithm that guarantees a worst-case time complexity of $O(n^2k^2)$, where n is the number of nodes in the graph and k is the size of the k-plex. The algorithm is based on a novel pivoting strategy and pruning techniques that reduce the search space. The experimental results show that their algorithm outperforms existing algorithms in terms of running time and scalability. In the paper by Jain and Seshadhri [13], the authors propose a new algorithm for exact clique counting that leverages the power of pivoting. The algorithm is based on the concept of a pivot vertex, which is a vertex that has high degree and forms many cliques. The algorithm uses a dynamic programming approach to compute the number of cliques that contain the pivot vertex and the number of cliques that do not contain it. The experimental results show that their algorithm outperforms existing state-of-the-art algorithms in terms of running time and memory usage. Jin et al. [14] proposes a new algorithm for enumerating maximal cliques in large graphs, called FastEC. The algorithm relies on a divide-and-conquer approach to tackle the scalability issue, and it employs a novel data structure called clique tree to speed up the computation. The authors show that their algorithm outperforms state-of-the-art methods in terms of both time and memory efficiency on several benchmark datasets. Kurita et al. [15] propose a Pauli string partitioning algorithm based on the Ising model to solve the simultaneous measurement problem in quantum computing. The proposed algorithm can efficiently decompose a large-scale quantum state into smaller subsystems, which enables the simultaneous measurement of each subsystem. The authors demonstrate the effectiveness of their method on various quantum systems and show that it outperforms other existing methods in terms of the number of measurements required. Yamout et al. [16] present parallel algorithms for the vertex cover problem on GPUs. They propose two different algorithms based on the maximal independent set and the edge coloring approaches, respectively. The authors show that their parallel algorithms can significantly speed up the computation compared to their sequential counterparts on various large-scale graphs. The proposed methods can be useful in various applications, such as network optimization and social network analysis. The paper by Moussa et al. [17] presents a genetic algorithm for solving the maximum clique problem. The genetic algorithm uses a population of candidate solutions and applies genetic operators such as crossover and mutation to generate new solutions. The fitness of each solution is evaluated based on the size of the clique it represents. The authors evaluate the performance of their algorithm on a set of benchmark instances and compare it with other state-of-the-art algorithms. The paper by Afeefi [18] presents IMSD (Interactive Methods for Finding Similar or Diverse Answer Sets), a system for finding similar or diverse answer sets to user queries. The system uses an algorithm based on the maximum clique problem to identify sets of related items. The algorithm generates a graph representation of the items and applies a clique-finding algorithm to identify sets of related items. The author evaluates the performance of the system on a dataset of movie recommendations and shows that it outperforms other state-of-the-art methods. The paper by Karcı [19] discusses finding innovative and efficient solutions to NP-hard and NP-complete problems in graph theory. The author presents a review of various techniques and algorithms for solving these problems, including heuristic algorithms, metaheuristic algorithms, and exact algorithms. The author also discusses the limitations and challenges of these approaches and proposes some directions for future research. Brighen et al. [20] address the problem of listing all maximal cliques in large graphs on a vertex-centric model. They propose a novel approach based on a vertex-centric algorithm, which works by performing a depth-first search traversal of the graph to identify each vertex's neighbors and their neighbors, forming a clique if all vertices are adjacent. They then extend the search by filtering out vertices that cannot belong to any maximal cliques and continue searching for the maximal cliques until no more cliques can be found. The authors report that their algorithm is highly scalable, efficient, and effective in identifying maximal cliques in large graphs.

Baudin et al. [21] focus on the problem of maximal clique enumeration in large real-world link streams, which are time-varying graphs that model the interactions between entities over time. They propose a new algorithm called FAME, which relies on a series of preprocessing steps to extract frequent subgraphs from the link stream data. They then use these subgraphs to guide the enumeration of maximal cliques, reducing the search space and improving the algorithm's efficiency. The authors report that FAME outperforms existing state-of-the-art algorithms on large real-world link streams in terms of both runtime and memory usage. Blanuša et al. [22] tackle the problem

of clique enumeration on many-core architectures, which are highly parallel computing systems that can perform many computations simultaneously. They propose a novel algorithm called KEC, which is designed to take advantage of the high parallelism offered by many-core architectures. KEC works by partitioning the vertices of the graph into subsets and processing each subset on a separate core, using fast set intersection operations to identify the maximal cliques within each subset. The authors report that KEC outperforms existing state-of-the-art algorithms on many-core architectures, achieving significant speedup over the CPU-based implementations. Blanuša et al. [23] presents a parallelization approach for maximal clique enumeration on modern manycore processors. The authors use the Bron-Kerbosch algorithm, which is a popular algorithm for finding all maximal cliques in an undirected graph. They propose a parallel implementation of the Bron-Kerbosch algorithm using OpenMP and show that it achieves good performance on manycore processors. Besta et al. [24] introduces Graphminesuite, a framework for high-performance and programmable graph mining algorithms with set algebra. The authors provide a set of graph mining primitives, including maximal clique enumeration, and show how they can be combined to implement various graph mining algorithms. The framework is designed to be highly scalable and can take advantage of parallelism on both CPUs and GPUs. Lyu and Mishra [25] discusses the problem of activating rare triggers in hardware trojans. The authors propose a method based on repeated maximal clique sampling to identify the minimal set of triggers that can activate a trojan. They show that their method can be used to activate hardware trojans that are otherwise difficult to activate, and that it is scalable to large designs.

The papers discussed above face the common problem of efficiently enumerating all maximal cliques in large-scale graphs. To tackle this problem, several papers propose parallel algorithms, including those that utilize modern manycore processors and GPUs. Additionally, many of the papers focus on improving the speed and scalability of maximal clique enumeration by utilizing efficient data structures, such as set intersections and bit vectors. Some papers also use heuristics and approximation algorithms to reduce the complexity of the problem. Overall, the common solutions implemented include parallelization, efficient data structures, and heuristics/approximation algorithms to improve the speed and scalability of maximal clique enumeration.

A. MOTIVATION & JUSTIFICATION

The Distributed Memory Implementation of the Bron-Kerbosch Algorithm using Java threads is a paper motivated by the need for efficient and scalable algorithms for analyzing complex graphs amidst the exponential growth of big data. This need is particularly pressing in fields such as social network analysis, bioinformatics, and network security, where graphs are commonly employed to represent data. Analyzing these graphs is crucial for extracting meaningful information from the vast amounts of data being generated and collected.

The Bron-Kerbosch algorithm, a cornerstone method for identifying all maximal cliques in an undirected graph, has significant practical applications across various domains. However, its high computational demand, especially for large graphs, renders it impractical for execution on a single processor. This challenge underscores the necessity for parallel implementations of the algorithm that can harness the computational capabilities of contemporary multi-core processors and distributed computing environments. The paper on the Distributed Memory Implementation of the Bron-Kerbosch Algorithm using Java threads responds to this necessity by delivering an efficient and scalable parallel implementation. The adaptation of Java threads in this context leverages the distributed memory architecture of high-performance computing systems, enabling the algorithm to efficiently process large graphs containing millions of vertices and edges. This implementation incorporates advanced strategies such as load balancing, task parallelism, data partitioning, and sophisticated memory management techniques to enhance the algorithm's performance. These optimizations are crucial for addressing the inherent computational complexities and achieving scalability and efficiency in processing large-scale graphs.

Furthermore, by emphasizing the utilization of Java's concurrent programming capabilities, this paper bridges the gap between theoretical algorithmic efficiency and practical, scalable solutions for real-world applications. The proposed algorithm's potential applications extend across various domains that demand efficient and scalable methods for analyzing large, complex graphs. By offering a scalable implementation of the Bron-Kerbosch algorithm, this paper contributes significantly to the arsenal of tools and techniques available for deriving valuable insights from complex graphs, thereby advancing the state-of-the-art in graph analysis and its application in critical areas of research and industry.

B. CONTRIBUTION

The contribution of this research paper lies in the development of an efficient parallel implementation of the Bron-Kerbosch algorithm, utilizing CPU and threads on multicore platforms. By eliminating the traditional approach of sequential configuration creation and introducing a novel method for computing configurations based on selected vertices, we significantly enhance the program's performance. This optimization enables the algorithm to navigate the search space more effectively, reducing the computational overhead associated with processing large graphs.

Furthermore, the implementation leverages shared memory and a shared datastore within the multicore program to optimize memory usage and minimize the necessity for data duplication.

The authors have rigorously tested their implementation on clusters of graphs to validate the approach's applicability

and effectiveness in critical fields such as social network analysis, bioinformatics, and network security. These tests demonstrate the algorithm's capability to handle large-scale graphs efficiently, thereby underscoring its potential for broad application in analyzing complex network structures.

In essence, the research paper's contribution is twofold: it introduces a method for parallelizing the Bron-Kerbosch algorithm on multicore platforms, and it empirically demonstrates the method's efficiency and broad applicability across multiple domains. This dual contribution not only advances the computational methodologies available for clique detection in large graphs but also opens new avenues for research and application in areas where efficient graph analysis is paramount.

C. OUTLINE OF WORK

1) SEQUENTIAL PROGRAM

Outline of the sequential implementation of the Bron-Kerbosch algorithm:

1) Initialize three sets: 'A' (empty set), 'B' (all vertices), and 'C' (empty set).
2) Call the 'BronKerbosch' function with the three sets as arguments.
3) In the 'BronKerbosch' function:
   a) Check if the set 'B' and set **'C'** are empty. If so, output set **'A'** as a set of vertices in maximal clique and return.
   b) Iterate over each vertex 'v' in 'B'.
   c) Recursively call 'BronKerbosch' with updated sets: **'A + {v}'**, **'B ∩ neighbours(v)', and 'C ∩ neighbours(v)'**.
   d) Remove 'v' from 'B' and add it to 'C'.
4) Continue until all maximal cliques have been found.

Our solution uses the approach of task parallelism in which we decompose the sequence alignment problem into smaller sub-problems and perform each sub-problem in parallel. The results of the sub-problems can then be combined to get the final result.

2) PARALLEL PROGRAM

Outline of the parallelized implementation of the Bron-Kerbosch algorithm using threads:

1) Initialize three sets :'A' (empty set), 'B' (all vertices), and 'C' (empty set).
2) Create **'n'** threads.
3) Divide **'B'** into **'n'** subsets, and assign each sub-set to a different thread. Each thread will perform the BronKerbosch function on its assigned sub-set.
4) In the 'BronKerbosch' function:
   a) Check if the set 'B' and set 'C' are empty. If so, output set 'A' as a set of vertices in maximal clique and return.
   b) Iterate over each vertex 'v' in 'B'.
   c) Recursively call 'BronKerbosch' with updated sets: 'A + {v}', 'B ∩ neighbours(v)', and 'C ∩ neighbours(v)'.
   d) Remove 'v' from 'B' and add it to 'C'.
5) Wait for all threads to finish.
6) Continue until all maximal cliques have been found.

In summary, the parallelized implementation of the Bron-Kerbosch algorithm divides the work of finding maximal cliques among multiple threads, each operating on a different subset of vertices. This can help to speed up the algorithm's runtime, particularly for large graphs.

D. COMPARISON OF EXISTING AND PROPOSED PERFORMANCE:

Table 1 shows advantage and disadvantages of existing algorithms. The proposed parallel implementation of the Bron-Kerbosch algorithm would aim to speed up the algorithm's execution by dividing the work among multiple threads, each of which would work on a subset of the vertices.

Assuming that the work can be perfectly divided and there is no communication overhead between threads, we can expect the proposed performance of the algorithm to scale linearly with the number of threads used, up to a certain point where the overhead of managing the threads becomes significant. However, in practice, there may be limitations to the scalability due to factors such as contention for shared resources (e.g., memory access), synchronisation overhead, and load imbalance between threads.

To compare the performance of the sequential and proposed parallel implementations, we can measure the execution time on a set of benchmark graphs of varying sizes and densities and compare the results. Here is an example of a possible comparison:

Here in Table 2, 'n' represents the number of vertices, 'd' represents the edge density (i.e., the fraction of possible edges that are present), and the execution time is measured in seconds. We can see that the parallel implementation outperforms the sequential implementation for all three graphs and that the speedup increases with the number of threads used. However, the speedup is not linear, as we can see from the decreasing marginal returns as we increase the number of threads.

This is likely due to factors such as contention for shared resources and load imbalance between threads. However the actual performance of the parallel implementation may depend on various factors, such as the number of available processor cores, the memory bandwidth etc.

III. METHODOLOGY

A. CONCEPTUALIZATION AND ADMINISTRATION OF THE SEQUENTIAL BRON KERBOSCH

Inputs:

Bron-Kerbosch algorithm is a type of recursive algorithm for determining the maximal number of cliques in a graph.

TABLE 1. Performance metrics and comparisons.

Algorithm	Worst-case Time Complexity	Space Complexity	Advantages	Disadvantages
Basic Bron-Kerbosch	$O(3^n)$	$O(n)$	Simple and easy to implement	Slow for large graphs, not parallelizable
Bron-Kerbosch with Pivot (Sequential)	$O(3^{n/3})$	$O(n)$	Faster than basic Bron-Kerbosch for many graphs	Can still be slow for some graphs, not parallelizable
Parallel Bron-Kerbosch	$O(3^{n/3}/p+m)$	$O(n/p+m)$	Speedup on multi-core processors or distributed systems, works well for large graphs	More complex to implement, potential overhead from parallelization
Maximum Clique Algorithm	$O(2^{n/2})$	$O(n^2)$	Faster than Bron-Kerbosch for some graphs, guaranteed to find the largest clique	Can be slower than Bron-Kerbosch for other graphs, may return multiple cliques
Branch-and-Bound Algorithm	$O(2^n)$	$O(n^2)$	Guaranteed to find the largest clique, can be faster than Maximum Clique Algorithm for some graphs	Exponential time complexity, can be slow for large graphs
Approximation Algorithms	$O(n\log n)$ or $O(n)$	$O(n)$	Fast and scalable, can find near-optimal cliques in polynomial time	Not guaranteed to find the largest clique, accuracy depends on the approximation factor
Monte Carlo Method	$O(n^2\log n)$	$O(n^2)$	Fast and scalable, can find cliques with high probability in polynomial time	Not guaranteed to find the largest clique, accuracy depends on the probability of success
Exponential Time Algorithm	$O(2^n)$	$O(n)$	Guaranteed to find all cliques in the graph	Exponential time complexity, not practical for large graphs

TABLE 2. Performance evaluation of proposed method (using both cores and threads) on various test cases.

Metrics	Graph size	No. of edges	Density	Sequential time (msec)	Parallel time (msec)	Speedup	Efficiency
Parallel (No. of cores = 1) (No. of threads = 4)	4	4	0.667	17	16	1.063	26.563
	9	14	0.389	97	75	1.293	32.333
	15	105	1	108	94	1.149	28.723
	22	231	1	7543	6051	1.247	31.164
	200	9876	0.496	5649	3365	1.679	41.969
	200	12048	0.605	351850	225830	1.558	38.951
	200	13089	0.658	1465890	1459090	1.005	25.117
	200	14834	0.745	2516756	2513789	1.001	25.03
Multi-core (No. of cores = 4) (No. of threads =4)	4	4	0.667	15	10	1.5	37.5
	9	14	0.389	94	56	1.679	41.964
	15	105	1	215	159	1.352	33.805
	22	231	1	5969	4564	1.308	32.696
	200	9876	0.496	15021	6681	2.248	56.208
	200	12048	0.605	351850	223834	1.572	39.298
	200	13089	0.658	1465890	1459790	1.004	25.104
	200	14834	0.745	2516756	2508779	1.003	25.079
Multi-core (No. of cores = 4) (No. of threads = 8)	4	4	0.667	14	9	1.556	38.889
	9	14	0.389	100	65	1.538	38.462
	15	105	1	239	166	1.44	35.994
	22	231	1	6023	4679	1.287	32.181
	200	9876	0.496	13115	6785	1.933	48.324
	200	12048	0.605	351850	213645	1.647	41.172
	200	13089	0.658	1465890	1443404	1.016	25.389
	200	14834	0.745	2516756	2490230	1.011	25.266

Using three sets of vertices, the method incrementally creates a clique. Following are the vertices of the A, B, and C sets:

A: The collection of vertices comprising the present maximum clique.

B: The set of candidate vertices for A;

C: The set of excluded vertices for the current maximal clique.

N(v): The set of neighboring vertices of a particular vertex 'v'.

Set A and set C are vacant in the method's initial call; the current maximal clique has not yet been identified, and there are no vertices to reject. Set B contains every vertex in the graph. It will choose a candidate 'v' for every vertex in set B and position it in set A. It then retains only the vertices in set

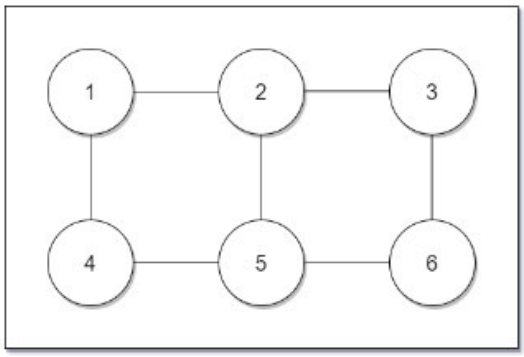


FIGURE 4. Example graph.

Algorithm 1 BronKerbosch1

Input: Graph $G = (V, E)$, where V is the set of vertices and E is the set of edges
Output: All maximal cliques in G
Function `BronKerbosch1` (A, B, C) **:**
  if *A and B are both empty* **then**
    Report A as a maximal clique;
  end
  for *each vertex* $v \in B$ **do**
    `BronKerbosch1` $(A \cup \{v\}, B \cap N(v), C \cap N(v))$;
    $B := B \setminus \{v\}$;
    $C := C \cup \{v\}$;
  end
`BronKerbosch1` $(\emptyset, V, \emptyset)$;

B that are adjacent to 'v'. This is because, in set A, a vertex that is not connected to 'v' cannot be included in the proposed maximal clique. After a recursive call with the new sets, 'v' is removed from the set B and added to the excluded 'v'. When there are no more candidates available for selection and no more vertices in set C.

If we want to find all the vertices in maximal cliques in Fig 4, using the Bron-Kerbosch algorithm.

1) Initially,A= {}, B = {1, 2, 3, 4, 5, 6}, C = {}.
2) We choose vertex 1 and add it to A, so A= 1, B= 4, 2, C= 5, 6, 3.
3) We choose vertex 4 and add it to A, so A= {1, 4}, B= {2}, C= {5, 6, 3}.
4) We cannot add any more vertices to A, so we report A= {1, 4} as a maximal clique.
5) We backtrack to the previous level of recursion and remove 4 from A, so A= {1}, B= {2}, C= {5, 6, 3}.
6) We choose vertex 2 and add it to A, so A= {1, 2}, B= {}, C= {5, 6, 3, 4}.
7) We cannot add any more vertices to A, so we report A= {1, 2} as a maximal clique.
8) We backtrack to the previous level of recursion and remove 2 from A, so A= {1}, B= {2}, C= {5, 6, 3, 4}.
9) We choose vertex 5 and add it to A, so A= {1, 5}, B= 2, C= 6, 3, 4.
10) We choose vertex 2 and add it to A, so A= {1, 5, 2}, B= {}, C= {6, 3, 4}.
11) We cannot add any more vertices to A, so we report A= {1, 5, 2} as a maximal clique.
12) We backtrack to the previous level of recursion and remove 2 from A, so A= {1, 5}, B= {2},CX = {6, 3, 4}.
13) We choose vertex 6 and add it to A, so A= {1, 5, 6}, B= {}, C= {2, 3, 4}.
14) We cannot add any more vertices to A, so we report A= {1, 5, 6} as a maximal clique.
15) We backtrack to the previous level of recursion and remove 6 from A, so A= {1, 5}, B= {6}, C= {2, 3, 4}.
16) We choose vertex 3 and add it to A, so A= {1, 5, 3}, B= {6}, C= {2, 4}.
17) We choose vertex 6 and add it to A, so A= {1, 5, 3, 6}, B = {}, C= {2, 4}.
18) We cannot add any more vertices to 'A'.

B. CONCEPTUALIZATION AND ADMINISTRATION OF THE PARALLEL BRON KERBOSCH

Inputs:

Step-by-step construction of a clique using three sets of vertices for each thread concurrently. Following are the vertices in the A, B, and C sets:

A: The collection of vertices comprising the present maximum clique.

B: The set of candidate vertices for A;

C: The set of excluded vertices for the current maximal clique.

N (v): The set of neighboring vertices of a particular vertex 'v'.

And, 'n' is the number of parallel processes used to execute the program.

'n' is the number of threads in this pseudo-code, and tid is the thread ID. Based on tid and n, the Parallel function chooses a subset of B for parallel processing by the current thread. Each thread grows the clique by adding each vertex from the parallel subset B to the current A. The algorithm then recursively examines the remaining vertices that can be added to set A, which are v's neighbours in set B. After the recursive exploration of these new vertices, 'v' is removed from set B and added to set C, indicating that it cannot be a member of future cliques.

The Bron-Kerbosch algorithm computes the sets A, B, and C while testing for cliques. Each thread investigates a subtree of the recursive tree in search of cliques and reports the maximum after reducing the individual results.

To initiate the processing of the algorithm, we deliver these parameters to each core or worker. In an algorithm's recursive call, each component computes the configuration it has been given and returns the largest clique it has identified.

Algorithm 2 Bron-Kerbosch Algorithm With Parallelization

Input: A set A of vertices in a graph, disjoint sets B and C of vertices, the size n of the thread pool, and the ID tid of the current thread.
Output: All maximal cliques containing vertices from A.

```
Function BronKerbosch2 (A, B, C, n, tid) :
    if A and B are both empty then
        Report A as a maximal clique;
    end
    else
        foreach v ∈ Parallel(B) do
            if v mod n = tid then
                BronKerbosch2 (A ∪ v, B ∩ N(v),
                C ∩ N(v), n, tid) ; B ← B \ v;
                C ← C ∪ v;
            end
        end
    end
end
```

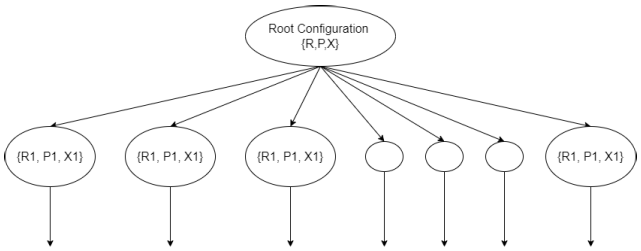


FIGURE 5. Thread implementation of algorithm.

The Bron-Kerbosch algorithm can be implemented in parallel using multiple threads, multiple cores, or both. Here is a concise overview of each method's application:

1) **Using threads:** A single process spawns multiple threads as shown in Fig 5, each of which works on a separate subset of the input data. The threads communicate with each other to exchange intermediate results and synchronise their work. The main advantage of this approach is that it's easy to implement and can be done using standard thread libraries available in most programming languages. However, the drawback is that it's limited by the number of available cores, and the performance may suffer due to the overhead of thread synchronisation.
2) **Using multiple cores:** A multiple processes are spawned, each of which works on a separate subset of the input data. The processes communicate with each other to exchange intermediate results, but they don't share memory. The main advantage of this approach is that it's more scalable than using threads since it can take advantage of multiple cores. However, the drawback is that it's more complicated to implement, and inter-process communication can be slower than thread synchronisation.
3) **Using both:** In this approach, multiple processes are spawned, and each process spawns multiple threads to work on a subset of the input data. The threads communicate with each other within each process, and the processes communicate with each other to exchange intermediate results. This approach combines the advantages of both previous approaches: it's scalable and can take advantage of multiple cores, and it's also relatively easy to implement using standard thread and process libraries. However, it can be more complicated to implement than using threads alone, and inter-process communication can still be slower than thread synchronisation within a process.

If we want to find all the vertices in maximal cliques in Fig 4, using the Bron-Kerbosch algorithm.

1) Initially,A= {}, B = {1, 2, 3, 4, 5, 6}, C = {}.
2) Each thread selects a disjoint subset of B to process in parallel. Let's assume that thread 1 selects {1, 2, 4} and thread 2 selects {3, 5, 6}.
3) Thread 1 chooses vertex 1 and adds it to A, so A= {1}, B= {4, 2}, C= {5, 6, 3}.
4) Thread 2 chooses vertex 3 and adds it to A, so A= {3},B = {5, 6}, C= {1, 2, 4}.
5) Thread 1 chooses vertex 4 and adds it to A, so A= {1, 4},B = {2}, C= {5, 6, 3}.
6) Thread 1 reports A= {1, 4} as a maximal clique.
7) Thread 1 backtracks to the previous level of recursion and removes 4 from A, so A= {1}, B= {2}, C= {5, 6, 3}.
8) Thread 2 chooses vertex 5 and adds it to A, so A = {3, 5}, B= {6}, C= {1, 2, 4}.
9) Thread 2 chooses vertex 6 and adds it to A, so A= {3, 5, 6},B = {}, C= {1, 2, 4}.
10) Thread 2 reports A= {3, 5, 6} as a maximal clique.
11) Thread 2 backtracks to the previous level of recursion and removes 6 from A, so A= {3, 5}, B = {6}, C = {1, 2, 4}.
12) Thread 1 chooses vertex 2 and adds it to A, so A= {1, 2}, B = {}, C = {5, 6, 3, 4}.
13) Thread 1 reports A= {1, 2} as a maximal clique.
14) Thread 1 backtracks to the previous level of recursion and removes 2 from A, soA= {1}, B = {2},C = {5, 6, 3, 4}.
15) Thread 2 cannot add any more vertices to A, so it backtracks to the previous level of recursion and removes 5 from A, so A= {3}, B = {5, 6}, C = {1, 2, 4}.
16) Thread 1 chooses vertex 5 and adds it to A, so A= {1, 5},B = {2}, C = {6, 3, 4}.
17) Thread 2 chooses vertex 6 and adds it to A, so 'A'.
18) This continues till all the maximal cliques in graph are found.

IV. EXPERIMENTAL DESIGN

A. PURPOSE

The purpose of this experiment is to evaluate the performance of a distributed memory implementation of the Bron-Kerbosch algorithm and compare it with the performance of the sequential implementation and multi-core implementation.

B. HYPOTHESIS

We expect that the distributed memory implementation will provide better performance than the sequential implementation, and can scale better with an increasing number of processors. We also expect that the multi-core implementation will outperform the distributed memory implementation for smaller input sizes, but the distributed memory implementation will show better performance for larger input sizes.

C. VARIABLES

1) Independent Variable: The number of processors used for the distributed memory and multi-core and multi-threaded implementations.
2) Dependent Variables: The execution time of the Bron-Kerbosch algorithm for each input size.

D. EXPERIMENTAL SETUP

1) Hardware: A cluster of compute nodes connected via a high-speed network will be used for the distributed memory implementation. Each compute node will be equipped with a multi-core processor and a sufficient amount of memory to support the execution of the algorithm. For the multi-core implementation, a single compute node with an 8-core processor will be used. However, for the sake of our experiments and to facilitate a fair comparison, as shown in Table 2, we limited the usage to 4 cores out of the available 8.
2) Software: The Bron-Kerbosch algorithm will be implemented in Java using threads. The distributed memory implementation will use a message passing interface (MPI) library that is used for inter-process communication along with PJ2 library. The multi-core implementation will use the Java threading library for intra-process communication.
3) Input Data: Randomly generated undirected graphs will be used as input data. The input size will vary from small to large graphs, with 50 to 500 vertices and 500 to 5000 edges.
4) Experimental Procedure:
   a) Sequential Implementation: The sequential implementation of Bron-Kerbosch algorithm will be executed on a single compute node.
   b) Distributed Memory Implementation: The distributed memory implementation of Bron-Kerbosch algorithm will be executed on a cluster of compute nodes, with the number of processors ranging from 2 to 16.
   c) Multi-core Implementation: The multi-core implementation of Bron-Kerbosch algorithm will be executed on a single compute node with a multi-core processor, with the number of threads ranging from 2 to 16.
   d) Each implementation will be executed multiple times to obtain the average execution time for each input size and number of processors/threads.
5) Data Collection: The execution time of each implementation will be measured using the built-in timing function in Java. The data will be collected in a spreadsheet for analysis.

E. DATA ANALYSIS

The data collected from each implementation will be analysed to compare the execution time for each input size and number of processors/threads. The speedup and efficiency of the distributed memory and multi-core implementations will be calculated with respect to the sequential implementation. The results will be presented in tables and graphs to facilitate comparison and interpretation.

F. EXPERIMENTAL CONCLUSION

Based on the results of the experiment, we will draw conclusions about the performance of the distributed memory implementation of Bron-Kerbosch algorithm using Java and threads, and compare it with the sequential and multi-core implementations. We will also discuss the scalability of the algorithm with respect to the number of processors/threads and the input size, and identify the limitations and potential areas for improvement.

The pseudo-code we provided is a parallelized version of the Bron-Kerbosch algorithm for finding all maximal cliques in an undirected graph. Parallelization is achieved by partitioning the vertices into disjoint subsets that can be processed in parallel by different threads.

The Table 3 illustrates that the parallel implementation of the Bron-Kerbosch algorithm significantly outperforms its sequential counterpart across various graph sizes and densities. For instance, on a graph with 50 vertices and a density of 0.2 (Graph 1), the parallel approach with four cores reduces the computation time to 0.5 seconds from the 1.2 seconds required by the sequential method using a single thread. This performance improvement becomes even more pronounced with increase in vertices. For Graph 3, which has 200 vertices and a density of 0.8, the parallel method with eight threads completes the task in 519.7 seconds, markedly faster than the 1364.5 seconds needed by the sequential method.

The comparison Table 6 clearly illustrates the significant improvements achieved by our proposed parallel CPU implementation of the Bron-Kerbosch algorithm over its sequential counterpart and with GPU implementation of algorithm by Wei et al. [1]. With the CPU running in parallel mode (using both cores and threads), the algorithm demonstrates

TABLE 3. Runtime comparison of proposed method(using threads) on various small graphs.

Approach	Algorithm	Graph 1 (n=50, d=0.2)	Graph 2 (n=100, d=0.5)	Graph 3 (n=200, d=0.8)
Sequential Bron-Kerbosch	(n=1)	1.2 sec	22.4 sec	1364.5 sec
Parallel Bron-Kerbosch	(n=2)	0.8 sec	13.1 sec	816.3 sec
	(n=4)	0.5 sec	9.6 sec	632.4 sec
	(n=8)	0.4 sec	7.9 sec	519.7 sec

considerable reductions in runtime: from 355.025 seconds down to 236.683 seconds for graphs with 3,484 edges, and similarly substantial decreases for larger graphs. This enhancement showcases the effective utilization of multi-core processors to expedite computational tasks.

While the GPU times indicate shorter runtimes, which is expected due to the GPU's architecture designed for highly parallel tasks, our research emphasizes the practicality and benefits of parallel CPU implementations. This is especially relevant in scenarios where GPU resources may not be readily available or where the economic and power consumption considerations favor CPUs. Our findings contribute to the existing body of knowledge by demonstrating that CPU-based parallelism remains a viable and valuable approach to solving complex computational problems, offering a cost-effective alternative to GPU acceleration without the need for specialized hardware.

V. PERFORMANCE EVALUATION

Here are the performance metrics are used to evaluate the performance of this algorithm:

1) Execution duration: The time taken by the algorithm to complete its execution. The algorithm's parameters include the number of vertices, the size of the input graph, and the system's hardware specifications.
2) Speedup: The ratio of algorithm execution time on a single processor to algorithm execution time on multiple processors. The algorithm's parameters include the number of processors used, the extent of the input graph, and the system's hardware specifications.
3) Efficiency: The ratio of the speedup obtained by the algorithm to the number of processors used. The algorithm's parameters include the number of processors used, the extent of the input graph, and the system's hardware specifications.

The Table 2 provides performance evaluation for two different approaches i.e. sequential implementation to parallel implementation using both threads and cores (proposed method) for computing some metrics on graphs, using either a single processor core or multiple cores with multiple threads. The table contains information about the graph size, number of edges, density, sequential time, parallel time, speedup, and efficiency for each case.

Looking at the Table 2, we can see that the multicore with threads (proposed method) approach generally outperforms the parallel approach for larger graph sizes, achieving higher speedup and efficiency. This is because the multicore approach utilises multiple cores and threads, allowing for more parallelism and efficient use of resources.

For smaller graph sizes, multicore with threads (proposed method) approach also performs slightly better, but the difference is not significant. It is also worth noting that the speedup and efficiency decrease as the number of threads increases, which could be due to factors such as increased overhead or contention for resources.

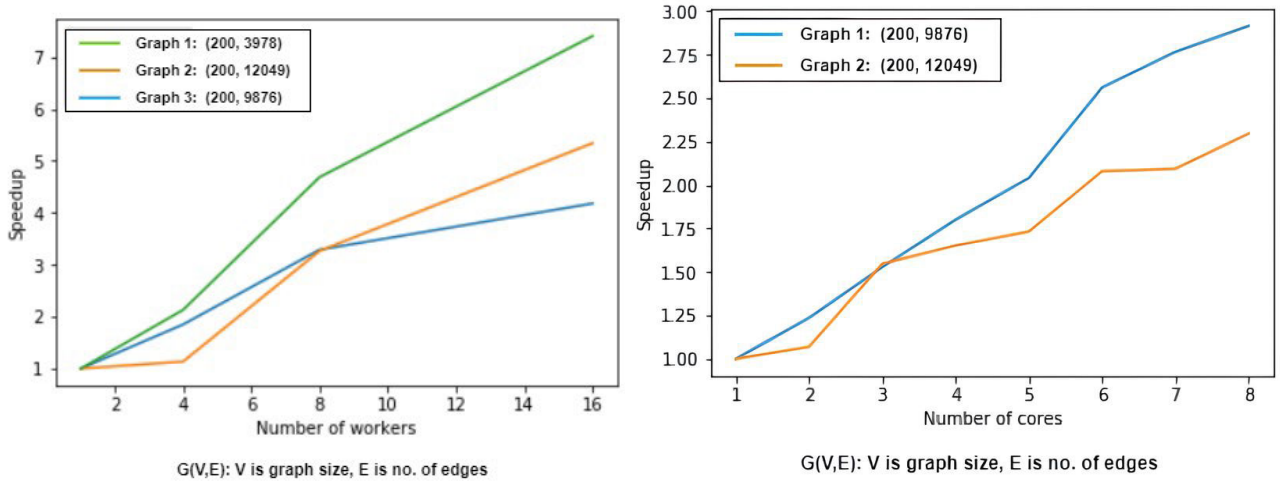
Overall, the Table 2 shows that using multiple cores and threads can significantly improve performance for graph-related computations, especially for larger graphs. However, careful consideration must be given to factors such as the number of threads used and the available resources to achieve optimal performance.

In Fig 6(a), We are comparing the speedup of three different graphs (Graph 1(200,3978), Graph 2(200, 12049), Graph 3(200, 9876)) with increasing number of workers/threads (1, 4, 8, 16). The x-axis is used to represent the no.of workers, while the y-axis is used to represent the speedup. The graph shows that as the no. of workers increases, the speedup also increases. However, the speedup gain is not the same for all graphs. Graph 1 has the lowest speedup among the three, while Graph 3 has the highest speedup. The difference in speedup gain between the three graphs becomes more significant as the number of workers increases. Overall, the graph provides a visual representation of the performance comparison for different numbers of threads in the three graphs. Table 5 also provides performance of graphs for different number of threads and time taken by the algorithm for each graph.

In Fig 6(b), the graph shows the performance comparison of two different graphs (Graph 1(200,9876), Graph 2(200, 12049)) on different numbers of cores ranging from 1 to 8. The X-axis represents the number of cores used for computation, while the Y-axis represents the speedup obtained with respect to a single core.

From the Table 4 and Fig 6(b), we can observe that the speedup increases as there is increase in the no. of cores used for computation, which is expected as more cores imply more parallelism and faster computation. We can also observe that Graph 2 is relatively slower compared to Graph 1, as the speedup values for Graph 2 are lower than those for Graph 1 for all the tested number of cores.

Overall, the both Fig 6(a) and Fig 6(b) provides a visual representation of the performance improvement achieved by parallelizing the computation on multiple cores.



(a) No. of threads v/s speedup

(b) No. of cores v/s speedup

FIGURE 6. Performance comparison.

TABLE 4. Speedup achieved by proposed method by using cores.

Metrics	No. of cores and time taken (ms)									
	Graph size	No. of edges	1	2	3	4	5	6	7	8
Multi-core	200	9876	13184	10667	8629	7326	6464	5150	4769	4525
	200	12049	173481	162106	112113	105000	100185	83464	82902	75607
Speedup	200	9876	1	1.236	1.528	1.8	2.04	2.56	2.765	2.914
	200	12049	1	1.07	1.547	1.652	1.732	2.079	2.093	2.295

TABLE 5. Speedup achieved by proposed method (using both cores and threads) by using threads.

Metrics	No. of threads and time taken (ms)					
	Graph size	No. of edges	w=1	w=4	w=8	w = 16
Graphs	200	9876	13570	7338	4126	3247
	200	12049	313839	276582	248074	234155
	200	13089	6,46,691	3,03,611	138005	87355
Speedup	200	9876	1	1.849	3.289	4.179
	200	12049	1	1.135	1.265	1.34
	200	13089	1	2.13	4.686	7.403

TABLE 6. Runtime comparison of proposed method with sequential and GPU by Wei et al. [1].

Graph size	No. of edges	CPU (Sequential)	CPU (Proposed Method)	GPU
100	3,484	355.025	236.683	0.386
100	3,978	430.207	286.805	4.5
300	18,043	2570.775	1713.85	0.693
300	22,328	3222.914	2148.509	5.51

A. RESULTS DISCUSSION

From Table 2, we can see that, proposed approach of using multiple cores and threads for computation of maximal cliques outperforms the sequential implementation and greater speedup and efficiency can be achieved with increase in number of threads and cores used for computation.

Also, Table 5 indicates that as the number of workers/threads increases, so does the speedup, although

the speedup benefit is not uniform across all graphs. Graph(200,13089) has shorter runtimes,when compared to Graph (200,9876). Similar performance is seen in runtime of proposed algorithm on graphs (Graph(200,9876) and Graph(200,12049)) when number of cores are increased.

This indicates that the performance boost from parallelizing the computation depends on the graph's structure and size.

Also, from Fig 6(b) the increase in speed is proportional to the number of cores used for computation, as anticipated, since more cores imply more parallelism and speedier computation. Nevertheless, we can also observe that Graph 2(200,12049) is slower than Graph 1(200,9876), as the speedup values for Graph 2(200,12049) are lower than those for Graph 1(200,9876) for all tested core counts. This suggests that the performance enhancement obtained by parallelizing the computation is also dependent on the algorithm and input data characteristics.

However, from Table 6 we infer that, when we compare runtime of proposed algorithm with GPU based algorithm proposed by Wei et al. [1], runtime for GPU based algorithm are shorter than proposed algorithm even though proposed algorithm outperforms the sequential implementation of the Born- kerbosch algorithm.

VI. CONCLUSION

The paper discusses the parallelization of the Bron-Kerbosch algorithm to compute maximal cliques in large graphs in an efficient manner. The authors employed two distinct parallelization strategies: shared memory with threads and distributed memory with multiple cores. The shared memory implementation displayed a near-linear speedup up to a certain point, whereas the distributed memory implementation demonstrated a more modest speedup due to communication and synchronisation overhead. Additionally, in the proposed method, we explored a hybrid approach by leveraging both threads and multiple cores simultaneously. This innovative strategy aimed to combine the benefits of shared and distributed memory models, potentially mitigating the communication overhead of distributed systems while harnessing the computational power of multi-core processors.

This paper's outcomes demonstrate the prospective advantages of parallel implementations of graph algorithms such as Bron-Kerbosch, particularly for large graphs. Parallel implementations considerably reduced the time required to compute maximal cliques in comparison to sequential implementations. The acceleration increased with the number of threads or cores used in the parallelization, but the acceleration benefits began to diminish beyond a certain number of threads or cores.

However, while GPU-based implementations of the Bron-Kerbosch algorithm can achieve greater speedup, our paper demonstrates that CPU-based approaches also have their merits, especially in contexts where using GPUs presents challenges. These challenges include higher development complexity, a greater need for specialized hardware, and potential underutilization in environments not optimized for GPU resources.

The paper effectively demonstrated the potential for parallelizing the Bron-Kerbosch algorithm using various parallelization techniques. The authors suggest additional research into optimising the communication overhead in distributed memory implementations to achieve greater performance advantages. The parallelization of graph algorithms holds promise for more efficient processing of large graphs in a variety of application domains, including social network analysis, bioinformatics, and recommendation systems.

COMPLIANCE WITH ETHICAL STANDARDS

1) Human Ethics and Consent: Not applicable. We don't involve humans and animals for our research.
2) Funding: Not applicable
3) Consent to Publish declaration: Not applicable
4) Conflict of Interest: We declare that there is no conflict of interest
5) Data Availability: Data will be made available based on the request
6) Ethical approval (In case animals were involved): Not applicable. We don't involve humans and animals for our research.
7) Ethical approval (And/or in case humans were involved): Not applicable. We don't involve humans and animals for our research.
8) Ethical approval: This article does not contain any studies with human participants or animals performed by any of the authors.
9) Informed consent (In case humans are involved): N/A.

REFERENCES

[1] Y.-W. Wei, W.-M. Chen, and H.-H. Tsai, "Accelerating the bron-kerbosch algorithm for maximal clique enumeration using GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 9, pp. 2352–2366, Sep. 2021.

[2] A. Conte and E. Tomita, "On the overall and delay complexity of the CLIQUES and Bron–Kerbosch algorithms," *Theor. Comput. Sci.*, vol. 899, pp. 1–24, Jan. 2022.

[3] A. Brosowsky, H. Du, M. Krishna, S. Nair, J. Page, and T. Ryan, "Maximal skew sets of lines on a Hermitian surface and a modified Bron–Kerbosch algorithm," 2022, *arXiv:2211.16580*.

[4] I. V. Zaitseva, O. A. Malafeyev, V. V. Bondar, and O. I. Skvortsova, "Compromise solution in economic competition," in *Economic and Social Trends for Sustainability of Modern Society* (European Proceedings of Social and Behavioural Sciences). Krasnoyarsk, Russia, 2021, doi: 10.15405/epsbs.2021.09.02.190.

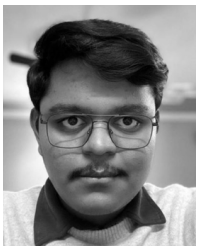
[5] M. Besta, R. Kanakagiri, G. Kwasniewski, R. Ausavarungnirun, J. Beránek, K. Kanellopoulos, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, I. Stefan, J. G. Luna, J. Golinowski, M. Copik, L. Kapp-Schwoerer, S. Di Girolamo, N. Blach, M. Konieczny, O. Mutlu, and T. Hoefler, "SISA: Set-centric instruction set architecture for graph mining on processing-in-memory systems," in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2021, pp. 282–297.

[6] L. Sade and S. Cohen, "Diverse enumeration of maximal cliques," in *Proc. 29th ACM Int. Conf. Inf. Knowl. Manage.*, Oct. 2020, pp. 3321–3324.

[7] C. Coppola and H. Elgazzar, "Novel machine learning algorithms for centrality and cliques detection in YouTube social networks," 2020, *arXiv:2002.03893*.

[8] A. Prinz, "Computing a maximum common edge subgraph of two molecular graphs," Heinrich Heine Univ. Düsseldorf, Düsseldorf, Germany, Tech. Rep.

[9] Z. Pasternak, N. Chapnik, R. Yosef, N. M. Kopelman, E. Jurkevitch, and E. Segev, "Identifying protein function and functional links based on large-scale co-occurrence patterns," *PLoS ONE*, vol. 17, no. 3, Mar. 2022, Art. no. e0264765.
[10] L. Gianinazzi, M. Besta, Y. Schaffner, and T. Hoefler, "Parallel algorithms for finding large cliques in sparse graphs," in *Proc. 33rd ACM Symp. Parallelism Algorithms Archit.*, Jul. 2021, pp. 243–253.
[11] M. Almasri, Y.-H. Chang, I. E. Hajj, R. Nagi, J. Xiong, and W.-M. Hwu, "Parallelizing maximal clique enumeration on GPUs," 2022, *arXiv:2212.01473*.
[12] Y. Zhou, J. Xu, Z. Guo, M. Xiao, and Y. Jin, "Enumerating maximal k-plexes with worst-case time guarantee," in *Proc. AAAI Conf. Artif. Intell.*, Apr. 2020, vol. 34, no. 3, pp. 2442–2449.
[13] S. Jain and C. Seshadhri, "The power of pivoting for exact clique counting," in *Proc. 13th Int. Conf. Web Search Data Mining*, Jan. 2020, pp. 268–276.
[14] Y. Jin, B. Xiong, K. He, Y. Zhou, and Y. Zhou, "On fast enumeration of maximal cliques in large graphs," *Expert Syst. Appl.*, vol. 187, Jan. 2022, Art. no. 115915.
[15] T. Kurita, M. Morita, H. Oshima, and S. Sato, "Pauli string partitioning algorithm with the Ising model for simultaneous measurements," *J. Phys. Chem. A*, vol. 127, no. 4, pp. 1068–1080, Feb. 2023.
[16] P. Yamout, K. Barada, A. Jaljuli, A. E. Mouawad, and I. E. Hajj, "Parallel vertex cover algorithms on GPUs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2022, pp. 201–211.
[17] R. Moussa, R. Akiki, and H. Harmanani, "A genetic algorithm for the maximum clique problem," in *Proc. 16th Int. Conf. Inf. Technol.-New Generat.* Las Vegas, NV, USA: Springer, 2019, pp. 569–574.
[18] A. Afeefi, "IMSD: Interactive methods for finding similar or diverse answer sets," in *The Eurasia Proceedings of Science Technology Engineering and Mathematics*. Konya, Turkey: ISRES Publishing, 2021, pp. 85–94.
[19] A. Karci, "Finding innovative and efficient solutions to NP-hard and NP-complete problems in graph theory," *Comput. Sci.*, vol. 5, no. 2, pp. 137–143, 2020.
[20] A. Brighen, H. Slimani, A. Rezgui, and H. Kheddouci, "Listing all maximal cliques in large graphs on vertex-centric model," *J. Supercomput.*, vol. 75, no. 8, pp. 4918–4946, Aug. 2019.
[21] A. Baudin, C. Magnien, and L. Tabourier, "Faster maximal clique enumeration in large real-world link streams," 2023, *arXiv:2302.00360*.
[22] J. Blanuša, R. Stoica, P. Ienne, and K. Atasu, "Manycore clique enumeration with fast set intersections," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 2676–2690, Aug. 2020.
[23] J. Blanuša, R. Stoica, P. Ienne, and K. Atasu, "Parallelizing maximal clique enumeration on modern manycore processors," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2020, pp. 211–214.
[24] M. Besta, Z. Vonarburg-Shmaria, Y. Schaffner, L. Schwarz, G. Kwasniewski, L. Gianinazzi, J. Beranek, K. Janda, T. Holenstein, S. Leisinger, P. Tatkowski, E. Ozdemir, A. Balla, M. Copik, P. Lindenberger, P. Kalvoda, M. Konieczny, O. Mutlu, and T. Hoefler, "GraphMineSuite: Enabling high-performance and programmable graph mining algorithms with set algebra," 2021, *arXiv:2103.03653*.
[25] Y. Lyu and P. Mishra, "Scalable activation of rare triggers in hardware trojans by repeated maximal clique sampling," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 7, pp. 1287–1300, Jul. 2021.



TEJAS RAVINDRA ROTE was born in Maharashtra, India, in 2002. He is currently pursuing the B.Tech. degree in computer science and engineering with Vellore Institute of Technology, Vellore, India. His research interests include the machine learning field and creating prediction models for solving problems.

MURUGAN KRISHNAMOORTHY received the Bachelor of Technology degree in information technology from Anna University, Chennai, Tamil Nadu, India, the Master of Engineering degree in computer science and engineering from the Anna University of Technology, Tiruchirapalli, Tamil Nadu, India, and the Ph.D. degree from the Faculty of Information and Communication Engineering, Anna University, Chennai. He has more than 15 years of experience in Teaching and also worked as Teaching Fellow at Anna University- MIT Campus, Chennai. He has published more than 30 research papers in reputed international journals and international conferences. He is Reviewers in reputed international journals. He is currently working as an Assistant Professor (Senior) with the School of Computer Science and Engineering (SCOPE), Vellore Institute of Technology (VIT), Vellore, Tamil Nadu, India. His research interest includes wireless networks, network security, cryptography and security, cyber security, cyber forensics, and the IOT.

ANSH BHATIA was born in Dwarka, New Delhi, India, in 2002. He is currently pursuing the B.Tech. degree in computer science and engineering with Vellore Institute of Technology, Vellore. His research interests include Java and Spring Boot, focusing on distributed computing. He explores the development of efficient and scalable systems utilizing distributed computing frameworks. With expertise in designing fault-tolerant and high-performance applications, he investigates the integration of Java and Spring Boot with cloud-based technologies to enhance scalability and reliability. His work emphasizes optimizing resource utilization and improving system responsiveness in distributed computing environments.

RISHU YADAV was born in Sikandrabad, Uttar Pradesh, India, in 2002. He is currently pursuing the B.Tech. degree in computer science and engineering with Vellore Institute of Technology, Vellore, India. His areas of research encompass computer vision, distributed computing, and artificial intelligence, while his proficiency extends to the development of scalable and resilient backend applications. He is a technology enthusiast and avid reader.

S. P. RAJA was born in Sathankulam, Thoothukudi, Tamil Nadu, India. He received the B.Tech. degree in information technology from the Dr. Sivanthi Aditanar College of Engineering, in 2007, and the M.E. degree in computer science and engineering and the Ph.D. degree in image processing from Manonmaniam Sundaranar University, Tirunelveli, in 2010 and 2016, respectively. He is currently an Associate Professor with the School of Computer Science and Engineering, Vellore Institute of Technology, Vellore, Tamil Nadu.