Parallelization of Heirholzer's Algorithm using OpenMP, MPI and CUDA

Devadarshini M
Department of Computer Science and Engineering
PSG Institute of Technology and Applied Research
Coimbatore, India
devadarshini10@gmail.com

Vaishnavi S
Department of Computer Science and Engineering
PSG Institute of Technology and Applied Research
Coimbatore, India
vaishnavi1731@gmail.com

R Manimegalai
Department of Computer Science and Engineering
PSG Institute of Technology and Applied Research
Coimbatore, India
drrm@psgitech.ac.in

Noor Mahammad SK
Department of Computer Science and Engineering
Indian Institute of Technology, Design and Manufacturing
Kancheepuram, India
noor@iiitdm.ac.in

2025 International Conference on Next Generation Computing Systems (ICNGCS) | 979-8-3315-9116-8/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICNGCS64900.2025.11183273

***Abstract* — Graphs are used practically in many different areas, such as transportation networks, road planning, and DNA sequencing. They are effective models for representing intricate linkages and streamlining numerous operations in these fields. Breaking down jobs into smaller, concurrent processes is known as parallelization, and it is a crucial strategy for increasing the effectiveness of graph algorithms. Breaking the big-problem into smaller sub-problems drastically cuts down on calculation time and improves system performance. Several methods have been proposed to parallelize graph algorithms. Two prominent APIs, OpenMP and MPI, are prominently used for parallization. OpenMP is suitable for shared memory systems, because it enables effective parallelization by taking advantage of several CPU cores. On the other hand, MPI works well with distributed memory systems and enables the distribution of computations across numerous connected workstations. In addition, NVIDIA GPUs benefit greatly from the parallelization of graph algorithms by CUDA. A powerful option for computationally heavy jobs, GPUs excel at parallel processing because of their large number of cores. The Heirholzer algorithm, which is the focus of this work, is parallized to address problems with Eulerian circuits and pathways in graphs. This work compares the performance of OpenMP, MPI, and CUDA and it is observed that the performance of Heirholzer's algorithm is enhanced in practical applications when parallelized.**

***Index Terms*—Graph Algorithms, Heirholzer's Algorithm, Eulerian circuit, Parallelization, OpenMP, MPI, CUDA**

I. Introduction

A. About Heirholzer's Algorithm

Heirholzer's algorithm is an effective approach for solving Eulerian circuits, a key idea in graph theory, and it employs a backtracking strategy. When backtracking is applied to a graph, it consistently finds an Eulerian circuit, which is a closed path that passes through every edge exactly once and so effectively covers every vertex. Backtracking iteratively probes unexplored edges while ensuring that each edge appears only once in the final circuit. A chosen vertex in the graph serves as the starting point for the Heirholzer algorithm execution phase. From here, the program investigates an unexplored edge and adds it to the circuit's expanding collection of edges.

Following a route that respects the graph's connection while being careful not to go back over an already-traveled edge, this exploration is continued. The algorithm's intelligence rests in its capacity to retrace its steps once it reaches a vertex with no more unexplored edges, making sure that it explores all potential paths until it visits every vertex and forms a circuit. A distributed algorithm for constructing an Eulerian tour in a network, focusing on efficient communication and time complexity is introduced in [1]. The algorithm's performance is characterized by a message and time unit complexity of $(1+r)(|E| + |V|)$, where $|E|$ is the number of communication links and $|V|$ is the number of nodes, with r depending on the network topology and traversal path. A modification to the algorithm enables the construction of a noncyclic tour with a slightly higher message and time complexity of $(1+r)(|E| + 2|V|)$. Real- time applications for this approach include shortest road route detection, DNA sequencing, network routing, data processing, computer network analysis, and game design. The importance of Heirholzer's algorithm resides in its capacity to effectively optimize routes and resolve challenging connectivity issues. It serves as the foundation for numerous applications, highlighting the extensive application of graph theory and algorithms in resolving problems in a variety of real-world settings.

B. Parallelization Techniques

A qualitative and exploratory study on the use of Parallel Processing APIs such as CUDA, OpenCL, OpenMP, and MPI, across various industries and application domains is conducted in [2]. Their SWOT analysis identifies the strengths, weaknesses, opportunities, and threats associated with these APIs, highlighting their roles in computational finance, AI, machine learning, data science, and numerical analysis. The study also reviews how companies leverage these APIs for high-performance computing, offering

insights into their effectiveness in building parallel programs and solving domain-specific problems. These APIs are useful tools for parallelizing various programs since each one shines in a certain computing environment and serves a specific function. OpenMP , which stands for Open Multi-Processing, was created to make shared memory parallelism easier. It provides programmers with a set of environment variables, library functions, and compiler directives that make it simple to parallelize programs. OpenMP enables programmers to effectively use the processing capability of multi-core computers by introducing pragmas to identify parallel portions, loops, and data sharing. Because several processor cores can access and edit the same memory area in shared memory systems, it is well suited for these systems.

Message Passing Interface (MPI) is specifically designed for distributed memory systems, in which numerous processes work on various memory domains and communicate by sending and receiving messages. Multi-cluster, multi- supercomputer, and other distributed computing environments can all be used with MPI to build parallel programs. Applications requiring intensive levels of data sharing and coordination depend on its ability to manage communication between different memory locations. Compute Unified Device Architecture, or CUDA for short, is a parallel computing platform and programming style made exclusively for Graphical Processing Units (GPUs) by NVIDIA. It gives programmers the tools to use the enormous parallel processing power of GPUs to speed up computationally demanding tasks. A runtime library, a collection of tools for memory management, and effective data transfer between the CPU and GPU are also provided by CUDA. This makes it a formidable option for applications such as scientific simulations, deep learning, and image processing that can profit from the parallel processing capability of GPUs.

II. Literature Review

An improved technique for kernel-oriented, CUDA-based GPU parallel implementations of the Dijkstra's and BFS graph algorithms is presented in [3]. This work compares the performance of parallel programming paradigms, namely OpenMP on CPU clusters and CUDA on GPU clusters, with respect to graph traversal algorithms such as Breadth First Search (BFS) and Depth First Search (DFS). Experimental results reveal that though OpenMP has idle and inter-thread communication overhead on CPUs, CUDA performs significantly faster by 187 to 240 times over the implementation on CPUs. The findings are useful to the programmers in choosing CUDA on GPUs for efficient high-intensity graph computations. Parallelization of fundamental graph algorithms such as BFS, DFS, Prim's, Kruskal, and Dijkstra on GPUs using CUDA in order to exploit their massively multithreaded architecture in the direction of faster execution is discussed in [4]. The research demonstrates that treating GPUs as inexpensive co-processors significantly reduces computation time for operations involving millions of vertices and edges when compared to traditional methods. The comparative analysis emphasizes the efficiency of CUDA-enabled GPUs in graph traversal and analysis, highlighting their cost-effectiveness and computational power. The performance analysis of parallel programming paradigms such as OpenMP and CUDA on CPU-GPU clusters with BFS and DFS graph algorithms is presented in [5]. It is shown that OpenMP on CPUs introduces overheads such as idle time and excess computation, whereas CUDA on GPUs achieves a speed-up of 187 to 240 times over CPU implementations. The study provides valuable insights for developers in selecting optimal programming paradigms, which shows that CUDA is the most efficient for high-intensity graph computations. A solution for parallelizing the Traveling Salesman Problem (TSP) in [6] provides a detailed explanation of the comparison of reducing execution time by parallelizing TSP using CUDA and OpenMP. Best results for OpenMP implementation have been achieved with four threads on a quad-core system. The execution time to find the optimal or near- optimal solution is reduced by seven times when the solution is implemented in parallel using CUDA. The implementation of a genetic algorithm to solve the NP-hard Traveling Salesman Problem (TSP) with CUDA on NVIDIA GPUs is examined in [7]. Leveraging the GPU parallelism and shared memory, the CUDA-based implementation presented notable performance improvements when compared to the CPU-based sequential genetic algorithm. Their results illustrate the capability of CUDA to efficiently accelerate heuristic algorithms that feature intensive interactions in computationally complex problems such as TSP. The OpenMP tasking model proposed in OpenMP 3.0 to cope with the irregular parallelism of complex applications is investigated in [8]. The model's design objectives, main characteristics are discussed in [8]. Scalability is demonstrated by comparing it with current models on a broad set of applications. The results in [8] have shown the ability of the OpenMP tasking model which takes advantage of unstructured parallelism with significant flexibility and performance improvements. Comparative studies with complex networks conclude that the speed of parallel algorithms is less than that of non-parallel algorithms for MPI based implementation [9]. A parallel Euler sequence assembly algorithm to handle the computational problems in large-scale biological sequence assembly is proposed in [10]. This algorithm, through parallel computing, assembles the genome fragments produced by WGS sequencing efficiently, eliminating the group partitioning error. The experimental results show that the algorithm has polynomial complexity and can be run on various systems, thus showing improved performance and a solution to the repeat problem in genome assembly. A comparative performance study of parallel programming models, including OpenMP, hybrid models combining one or more of OpenMP, MPI, CUDA , applied to graph algorithms such as Floyd's and Kruskal's in presented in [11]. The findings in [11] demonstrate that GPU implementations outperform CPU implementations, with the hybrid [MPI+CUDA] framework yielding significant speedups for both algorithms i.e. 19.03x for Floyd's and 27.26x for Kruskal's when compared to OpenMP. The study underscores the efficiency of GPU clusters for optimizing complex scientific computations by effectively minimizing communication and computation overlap. A detailed investigation on the optimization of shortest path algorithms such as Floyd-Warshall which computes paths between all pairs, and Dijkstra's algorithm, which finds paths for a single source using OpenMP and MPI to parallelize it is discussed in [12]. These significant optimizations enable the operations such as networking or routing of very large graphs to leverage speedups in execution time. The work in [12] demonstrates the

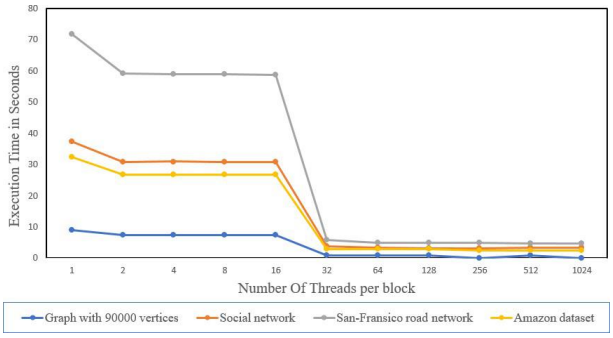

Figure 8. Number of Threads per Block vs. Execution Time using CUDA for Various Datasets

Heirholzer's algorithm is executed using CUDA for the four datasets as mentioned in Table 1.The CUDA implementation greatly reduced the execution times for all datasets. Optimal performance was achieved at specific threads per block: 128 threads for the Graph with 9000 vertices(0.89 seconds), 128 threads for the Social Network (3.17 seconds), 512 threads for the Amazon dataset (4.72 seconds), and 256 threads for the San Francisco Road Network (2.43 seconds). Compared to single-thread execution, CUDA delivered substantial speedups by leveraging parallelism. Increasing threads commonly increased performance to optimum level, beyond which performance would degrade slightly owing to resource contention.

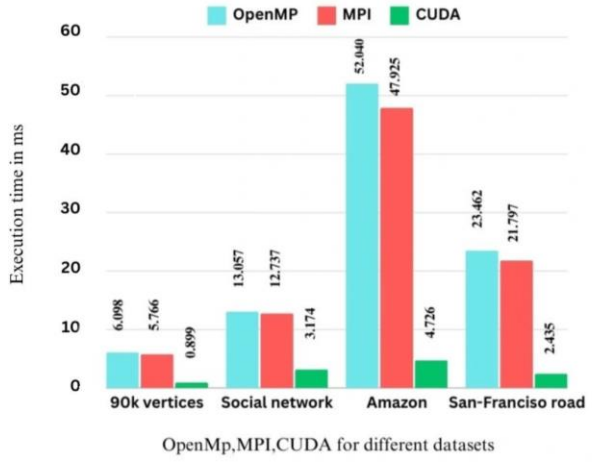

Fig. 9. Comparison based on Execution Time

The outcome of a careful examination of the data produced by the use of OpenMP, MPI, and CUDA is shown in Fig 9. This serves as a comparison of three various application programming interfaces (APIs). A persuasive conclusion is drawn from the data in Fig 9 indicating CUDA excels in offering an optimal and efficient solution, outperforming both OpenMP and MPI. This conclusion strongly recommends CUDA while looking for the best performance in comparable computational jobs

IV. Conclusions

A comparative analysis of parallelization approaches using OpenMP, MPI, and CUDA for Heirholzer's Algorithm is presented in this work. This provides a clear view of the strengths and weaknesses of each parallelization API. OpenMP is specifically designed for shared memory systems, thus it is an efficient mechanism for parallelizing loops, which is very helpful in dealing with small datasets. The reduction clause and logical operations enable parallel computation with significant time saving, as evident with optimal thread counts. MPI is, however, very efficient for use in distributed memory systems due to the provision of process-level parallelism that affords scalable computation on the node. Indeed, this shows that more threads are efficient in partitioning large data sets. CUDA, however, shows the highest performance gains because of its fine- grained parallelism and the ability to exploit the architecture of the GPU. This method of using threads and blocks to distribute computations efficiently among thousands of cores makes it the best choice for computationally intensive tasks. Comparative results, supported by the analysis of execution time on different datasets, show that CUDA always outperforms both OpenMP and MPI for Heirholzer's algorithm. As such, it is the best API for high-performance applications requiring maximum efficiency and scalability.

References

1. S. A. M. Makki, "A distributed algorithm for constructing an Eulerian tour," 1997 IEEE International Performance, Computing and Communi- cations Conference, Phoenix/Tempe, AZ, USA, 1997, pp. 94-100, doi: 10.1109/PCCC.1997.581385.
2. A., Shajil Kunte, Srinivasa. (2023). SWOT Analysis of Parallel Pro- cessing APIs - CUDA, OpenCL, OpenMP and MPI and their Usage in Various Companies. International Journal of Applied Engineering and Management Letters. 300-319. 10.47992/IJAEML.2581.7000.0206.
3. Ganjan single, Amirta Tiwari and Dhirendra Pratap Singh, "New Ap- proach for Graph Algorithms on GPU using CUDA", International Journal of Computer Applications Maulana Azad National Institute of Technology Bhopal, June 2013.
4. Chetan D. Pise and Shailendra W. Shende, "Parallelization of Graph Algorithms on GPU Using CUDA", International Journal of Electrical Electronics and Computer Systems by Dept. of Information Technology Yeshwant raoChavan College of Engineering, January 2014.
5. B. N. Chandrashekhar, H. A. Sanjay and T. Srinivas, "Performance Analysis of Parallel Programming Paradigms on CPU-GPU Clusters," 2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS), Coimbatore, India, 2021, pp. 646-651, doi: 10.1109/I- CAIS50930.2021.9395977.
6. R. Saxena, M. Jain, S. Bhadri and S. Khemka, "Parallelizing GA based heuristic approach for TSP

over CUDA and OPENMP," 2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Udupi, India, 2017, pp. 1934-1940, doi: 10.1109/ICACCI.2017.8126128.
7. Chen, S., Davis, S., Jiang, H., Novobilski, A. (2011). CUDA-Based Genetic Algorithm on Traveling Salesman Problem. In: Lee, R. (eds) Computer and Information Science 2011. Studies in Computational Intelligence, vol 364. Springer, Berlin, Heidelberg.
8. E. Ayguade et al., "The Design of OpenMP Tasks," in IEEE Transactions on Parallel and Distributed Systems, vol. 20, no. 3, pp. 404-418, March 2009, doi: 10.1109/TPDS.2008.105.
9. Predari, M., Tzovas, C., Schulz, C., Meyerhenke, H. (2021). An MPIbased Algorithm for Mapping Complex Networks onto Hierarchical Architectures. In: Sousa, L., Roma, N., Tom´as, P. (eds) Euro-Par 2021: Parallel Processing. Euro-Par 2021. Lecture Notes in Computer Science(), vol 12820. Springer, Cham.
10. Wei Shi and Wanlei Zhou, "A parallel Euler approach for largescale biological sequence assembly," Third International Conference on Infor- mation Technology and Applications (ICITA'05), Sydney, NSW, 2005, pp. 437-441 vol.1, doi: 10.1109/ICITA.2005.41.
11. Chandrashekhar, B.N., Sanjay, H.A. (2019). Performance Study of OpenMP and Hybrid Programming Models on CPU–GPU Cluster. In: Shetty, N., Patnaik, L., Nagaraj, H., Hamsavath, P., Nalini, N. (eds) Emerging Research in Computing, Information, Communication and Applications. Advances in Intelligent Systems and Computing, vol 906. Springer, Singapore.
12. R. Awari, "Parallelization of shortest path algorithm using OpenMP and MPI," 2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), Palladam, India, 2017, pp. 304-309, doi: 10.1109/I-SMAC.2017.8058360.
13. J. Nickolls and W. J. Dally, "The GPU Computing Era," in IEEE Micro, vol. 30, no. 2, pp. 56-69, March-April 2010, doi: 10.1109/MM.2010.41.
14. Chanthini, P. & Shyamala, K.. (2016). A Survey on Parallelization of Neural Network using MPI and Open MP. Indian Journal of Science and Technology.9.10.17485/ijst/2016/v9i19/93835.
15. Al-Mulhem, Muhammed & Aidhamin, A. & Al-Shaikh, Raed. (2013).On benchmarking the matrix multiplication algorithm using OpenMP,MPI and CUDA programming languages. WMSCI 2013 - 17th World Multi-Conference on Systemics, Cybernetics and Informatics, Proceedings.1. 40-4