2024 International Conference on Smart Systems for Electrical, Electronics, Communication and Computer Engineering (ICSSEECC) | 979-8-3503-7817-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICSSEECC61126.2024.10649542

Performance Analysis of Parallelized PageRank Algorithm using OpenMP, MPI and CUDA

Visali V S
Department of CSE
PSG Institute of Technology and Applied Research
Tamilnadu, India
visali.vs.4903@gmail.com

Manimegalai R
Department of CSE
PSG Institute of Technology and Applied Research
Tamilnadu, India
drrm@psgitech.ac.in

Noor Mahammad S K
Department of CSE
Indian Institute of Information Technology, Design and Manufacturing
Tamilnadu, India
noor@iiitdm.ac.in

Sunitha Nandhini A
Department of CSE
PSG Institute of Technology and Applied Research
Tamilnadu, India
asn@psgitech.ac.in

***Abstract*— Web search engines have developed into a crucial tool for effectively and quickly locating information among the vast web data. The PageRank algorithm is essential for web search as it measures the importance and relevance of web pages based on their incoming links, allowing search engines to rank results by prioritizing high-quality and authoritative content. This ensures that users receive more accurate and valuable information. The PageRank algorithm improves search engine results by assigning a numerical weight to each element in a hyperlinked set of documents, effectively measuring the importance of web pages based on quantity and quality of links pointing to them. With billions of web pages and an extensive network of hyperlinks, the traditional sequential computation of PageRank becomes impractical for timely and efficient processing. Parallelization allows the algorithm to be distributed across multiple processors or computing nodes, enabling simultaneous computation of PageRank scores for different web pages. The main objective of this work is to parallelize the PageRank algorithm using OpenMP, MPI and CUDA and to compare their execution time to find the optimal one. OpenMP simplifies shared-memory parallelism, MPI facilitates communication between distributed processes and CUDA harnesses GPU power for high-performance parallel processing in diverse parallel computing environments. The experimental results demonstrate notable performance enhancements through parallelization using different technologies: OpenMP improves the algorithm's performance by 49.7%, MPI by 62.4%, and CUDA by 84.3%. Hence, optimal results are found when the PageRank algorithm is parallelized using CUDA.**

Keywords—PageRank Algorithm, Hyperlinks, Parallelization, OpenMP, MPI, CUDA, GPU.

I. OVERVIEW OF PAGERANK ALGORITHM

A. About the algorithm

Search engines use a link analysis algorithm called PageRank to assign a numerical weight to each element of a hyperlinked set of documents, such as the World Wide Web[6]. The algorithm is named after Larry Page, one of the co-founders of Google, and it was developed by him and Sergey Brin. The basic idea behind PageRank is to measure the importance of web pages based on the structure of the hyperlink graph. In essence, it views a link from page-A to page-B as a vote by page-A for page-B. The page with more hyperlinks receives more votes and has higher page-rank. Mathematically, PageRank can be computed using Equation (1).

$$PR(A) = (1-d) + d\left(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \cdots + \frac{PR(N)}{L(N)}\right) \quad (1)$$

In Equation (1), *PR(A)* is the PageRank of page A, *d* is a damping factor, usually set to 0.85, *PR(B)*, *PR(C)*, ..., *PR(N)* are the PageRanks of pages that link to page A, and *L(B)*, *L(C)*, ..., *L(N)* are the number of outbound links on pages B, C, ..., N.

The page-rank of a page is determined by adding together the pageranks of all the pages that are linked to it, and then dividing that total by the total number of outbound links on all those pages. The pagerank calculation is an iterative process, and it converges as the pageranks stabilize. The algorithm is used by search engines to order search results, with pages having higher pageranks considered more relevant.

B. Advantages and applications of PageRank algorithm

The PageRank algorithm has several advantages that contribute to its widespread application. One key strength lies in its resistance to manipulation and spam, ensuring a more authentic assessment of a page's importance based on genuine link relationships rather than artificial tactics. Moreover, PageRank offers a global perspective by considering the entire link structure of the web. This comprehensive approach provides a more accurate evaluation of a page's significance, moving beyond simplistic counting-based algorithms. Another notable advantage is the algorithm's adaptability. Beyond its original application in web search, PageRank can be seamlessly applied to various network types. This versatility extends its utility to domains such as social networks, citation networks, and recommendation systems, making it a robust choice for a wide range of applications. PageRank's scalability is a crucial feature, especially in the context of the vast datasets encountered on the web. The algorithm efficiently scales with the size of the network, enabling the analysis and ranking of large amounts of information without compromising performance. Turning to its applications, PageRank plays a pivotal role in search engine ranking. By evaluating the relevance and importance of web pages, it significantly enhances the accuracy of search results, thereby improving the overall user experience.

Beyond search engines, PageRank finds application in recommendation systems, where it identifies and suggests items based on user preferences and network structures, enhancing content recommendation on online platforms. In social network analysis, PageRank is instrumental in identifying influential nodes and communities within networks. This capability aids in understanding information flow and influence propagation in various social contexts. Moreover, in academic and research settings, PageRank is applied to citation networks, providing insights into the significance and impact of scholarly articles and aiding researchers in identifying influential works. PageRank's versatility extends to biology, where it is applied in network analysis to identify critical nodes in protein-protein interaction networks or gene regulatory networks. This application assists researchers in understanding key components within biological systems.

C. Need for parallelization

The sequential implementation of the PageRank algorithm, while effective for smaller graphs, faces significant challenges when dealing with the escalating scale and complexity of modern web graphs and networks [1]. As the size of these graphs grows exponentially, a sequential approach becomes computationally prohibitive, resulting in extended processing times and limitations in scalability. Parallelization allows for the distribution of the workload across multiple processors or nodes, enabling the algorithm to handle much larger graphs than in a sequential implementation. It accelerates the computation of PageRank scores by allowing for the concurrent execution of independent calculations. This reduction in computational time is especially vital in the context of real-time applications and dynamic graphs where timely updates of PageRank scores are crucial for accurate analysis and decision-making [7]. Hence parallelization of the PageRank algorithm is essential. In this work, the PageRank algorithm is parallelized using OpenMP, MPI and CUDA and the performance of the parallelized algorithm is analyzed. The steps involved in parallelizing any application are shown in Figure 1.

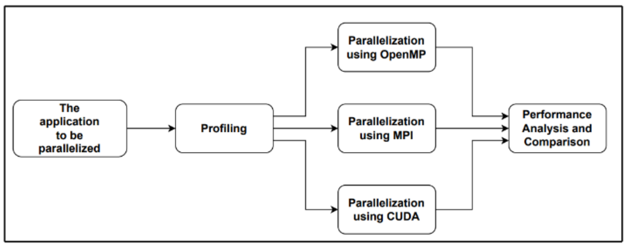

The application to be parallelized
Profiling
Parallelization using OpenMP
Parallelization using MPI
Parallelization using CUDA
Performance Analysis and Comparison

Fig. 1 Steps Involved in Parallelizing a Typical Application.

The process of parallelizing an application involves several key steps, each contributing to the efficient distribution of computational tasks across multiple processors or cores. Firstly, the application that needs to be parallelized is identified. This involves analyzing the codebase and identifying sections of the program that can be divided into parallel tasks. Following the identification of parallelizable sections, profiling the application becomes crucial. Profiling involves assessing the performance characteristics of the application to understand its resource utilization, bottlenecks, and areas where parallelization can yield the most significant improvements. Profiling tools help identify the parts of the code that consume the most computational resources, guiding the subsequent parallelization efforts. The actual parallelization process involves implementing parallel constructs in the code. OpenMP, MPI, and CUDA are three widely used parallelization frameworks. OpenMP is often suitable for shared-memory architectures, allowing developers to introduce parallelism through pragmas in the code. MPI is employed for distributed-memory architectures, enabling communication between different processors. CUDA, on the other hand, is specific to NVIDIA GPUs, allowing developers to offload parallelizable tasks to the GPU for execution. Once the parallelized versions of the application using OpenMP, MPI, and CUDA are developed, execution time comparison becomes a crucial step. This involves running the original, sequential version of the application alongside the parallelized versions, measuring the time each takes to complete a given task or set of tasks. This comparison provides insights into the effectiveness of parallelization, highlighting performance improvements achieved through concurrent processing.

D. Advantages of parallelization

Parallelizing an application offers several compelling advantages that significantly impact its performance and efficiency. One key benefit is the potential for improved speed and reduced execution time. By dividing tasks into parallel threads or processes that can execute simultaneously, parallelization harnesses the power of multiple processors or cores. This concurrent execution allows the application to process more data in less time, resulting in faster overall performance. Another advantage of parallelization is enhanced scalability. As the size of datasets or computational requirements grows, parallelized applications can effectively scale to meet increased demands. This scalability is particularly crucial in addressing the challenges posed by large-scale computing tasks, ensuring that the application remains efficient and responsive even when dealing with extensive and complex workloads. Parallelization also contributes to resource utilization, maximizing the use of available hardware resources. By distributing workloads across multiple processors, the application can make optimal use of the processing power, memory, and other resources, leading to more efficient utilization of the underlying hardware infrastructure.

E. Parallel Programming Paradigms

Although there are various parallelization techniques, this work focuses on implementing the PageRank algorithm using three APIs: OpenMP, MPI, and CUDA.

OpenMP: Open Multi-Processing (OpenMP) is an API that facilitates shared memory parallelism. It provides a list of environment variables, library functions, and compiler instructions. OpenMP allows programmers to parallelize their programs by providing pragmas to indicate loops, data sharing, and parallel portions. It supports nested parallelism, allowing parallel regions within other parallel regions. It also provides mechanisms for thread affinity, allowing control over thread placement on processors [13].

MPI: Message Passing Interface (MPI) is a message passing parallel programming protocol and library specification. It is designed for distributed memory systems, where several processes run on various memory domains and actively exchange messages with one other. MPI programs can benefit significantly from exploiting the underlying system's topology. While MPI is typically associated with static process allocation, modern MPI implementations offer dynamic process management [4].

CUDA: NVIDIA created the parallel computing platform and programming language known as Compute Unified Device Architecture (CUDA) for Graphics Processing Units (GPUs). It lets programmers take advantage of the massive parallel processing power of GPUs to speed up computationally intensive activities. CUDA supports dynamic parallelism, allowing a GPU thread to launch new threads. Though this feature is powerful, it comes with overhead, and its optimal use depends on the specific workload [15]. This work explores the necessity and benefits of parallelizing the PageRank algorithm using OpenMP, MPI and CUDA and identifies which API gives optimal performance.

II. LITERATURE REVIEW

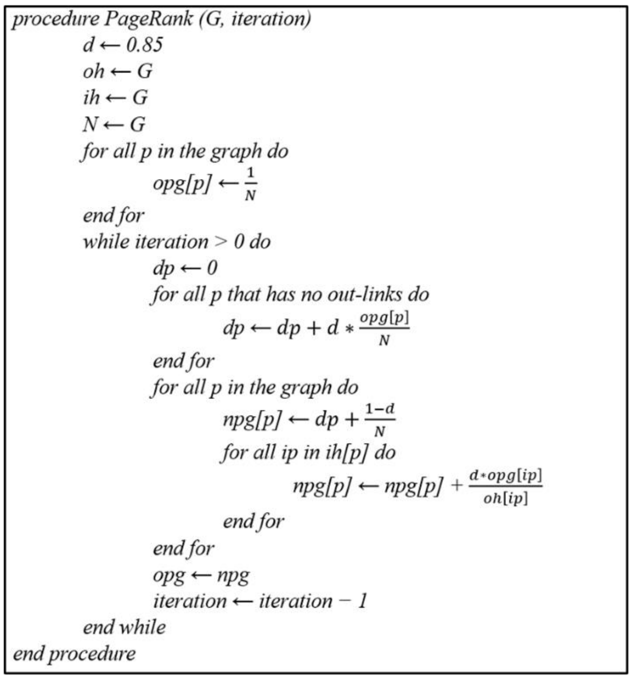
Christian Kohlschütter et al. have introduced the Gauß-Seidel iterative method, traditionally employed in numerical analysis for solving linear systems [1]. The proposed methodology in [1] exploits the inherent parallelism of web link structures, to enhance the convergence speed of PageRank algorithms. Kumar et al. have reported a 2.8 times speedup in the efficacy of CUDA in accelerating PageRank computations, signaling a potential shift toward more efficient and streamlined hardware configurations [2]. The findings presented by Cevahir et al. in [3] contribute significantly to optimize PageRank computations in large-scale web graph analyses. The integration of advanced sparse matrix partitioning models, coupled with innovative compression schemes and considerations for initially distributed data, showcases a nuanced approach to addressing the challenges posed by the ever-expanding web [3]. The contribution of Manaskasemsak et al. balances the benefits of MPI and multi-threading for large-scale PageRank computations [4]. Desikan et al. have introduced a novel approach to PageRank computation through a divide-and-conquer strategy, to enhance the efficiency of large graph ranking metrics [5]. Anakath et al. have leveraged the multidisciplinary nature of web ranking algorithms, encapsulating applied mathematics and computer science. The ongoing research landscape reflects a commitment to improve the efficiency of these algorithms, addressing challenges, and adapting to the evolving nature of the web [7]. Early work in parallel computing was introduced by Segel et al. in [10]. The potential of concurrent processor usage in solving computational problems is demonstrated in [10]. For instance, a problem with n independent steps and n processors can achieve a time complexity of $O(1)$ in parallel, when compared to $O(n)$ in a serial execution scenario. Yang et al. have contributed to the evolving field of parallel programming by using a hybrid approach that amalgamates CUDA, OpenMP, and MPI. This strategy addresses the challenges of parallel computation in heterogeneous GPU clusters, showcasing a thoughtful integration of programming models to maximize efficiency. As the field continues to evolve, hybrid approaches like the one proposed in this paper serve as valuable contributions to the arsenal of techniques for achieving optimal parallelization in diverse computing environments [8]. Mohamed et al. highlight the importance of Parallel computing using OpenMP, MPI, and CUDA. As the computing landscape continues to evolve, parallel computing stands out as a key paradigm for achieving improved computational efficiency and addressing the challenges of contemporary hardware limitations [9].

Duong et al. have addressed the pressing need for efficient web rank score computation through the innovative use of GPU parallelism for PageRank [11]. Zhou et al. have introduced an efficient parallel PageRank design based on an edge-centric scatter-gather model, aiming to overcome challenges related to poor locality and optimize memory performance [12]. Noaje et al. have provided a comparative analysis of two prominent models, namely, heavyweight MPI processes and lightweight OpenMP threads, in the context of multi-GPU environments. The focus on performance, implementation complexity, and optimization techniques such as pinned memory enhances our understanding of the trade-offs involved in adapting these models [14].

III. PARALLELIZATION OF PAGERANK ALGORITHM

A. Overview of parallelization of the algorithm

In this algorithm, interlinked web pages are represented as a directed graph. Each web page corresponds to a node in the graph, and hyperlinks between pages are represented as directed edges. The graph is typically a directed graph since hyperlinks have a specific direction, pointing from one page to another. Graph algorithms are used to model the relationships between web pages and calculate the PageRank scores. The primary reason for converting web pages into a graph is to capture the inherent structure and connectivity of the web. By representing the web as a graph, the algorithm can analyze the link structure and quantify the importance of each webpage based on its connections. The PageRank algorithm assigns importance scores to web pages based on the link structure of the Internet. It iteratively redistributes scores by considering both incoming and outgoing links until convergence. The pseudocode for the algorithm is presented in Figure 2 [6]. The implemented PageRank algorithm consists of five functions which are *initializeGraph( ), addLink( ), generateRandomGraph( ), pageRank( )*, and the *main( )* function. The *initializeGraph( )* function initializes the graph with zero counts and allocates memory for inlinks pages. The *addLink()* function adds a directed link from one vertex to another, reallocating memory as needed. The *generateRandomGraph()* function creates a random directed graph with a 50% probability of adding a link between vertices. The *pageRank( )* function computes PageRank scores iteratively for each vertex over a specified number of iterations. Finally, the *main()* function executes the program, measures computation time, and frees allocated memory for inlinks pages.

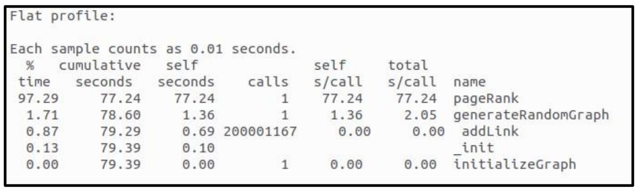


```
procedure PageRank (G, iteration)
        $d \leftarrow 0.85$
        $oh \leftarrow G$
        $ih \leftarrow G$
        $N \leftarrow G$
        for all p in the graph do
                $opg[p] \leftarrow \frac{1}{N}$
        end for
        while iteration > 0 do
                $dp \leftarrow 0$
                for all p that has no out-links do
                        $dp \leftarrow dp + d * \frac{opg[p]}{N}$
                end for
                for all p in the graph do
                        $npg[p] \leftarrow dp + \frac{1-d}{N}$
                        for all ip in ih[p] do
                                $npg[p] \leftarrow npg[p] + \frac{d * opg[ip]}{oh[ip]}$
                        end for
                end for
                $opg \leftarrow npg$
                $iteration \leftarrow iteration - 1$
        end while
end procedure
```

Fig. 2 Pseudocode of PageRank algorithm.

B. Identification of Hotspots in PageRank Algorithm using Profiling

Profiling a program involves analyzing its execution behavior to identify performance bottlenecks, memory usage, and other characteristics. Profiling provides insights into how much time is spent in different parts of the code, which functions are called frequently, and how much memory is allocated. Profiling is crucial for identifying areas where improvements can be made, ultimately leading to better program performance. While various tools are available for profiling, in this work *gprof* of the GNU Compiler Collection (GCC) is used [16]. This generates a flat profile which lists functions and their execution time as shown in Figure 3. Profiling is helpful in identifying hotspots in *pageRank( )* function. Optimizing this function would result in overall reduction in execution time.



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 97.29     77.24    77.24        1    77.24    77.24  pageRank
  1.71     78.60     1.36        1     1.36     2.05  generateRandomGraph
  0.87     79.29     0.69 200001167     0.00     0.00  addLink
  0.13     79.39     0.10                             _init
  0.00     79.39     0.00        1     0.00     0.00  initializeGraph
```

Fig. 3 Identification of Hotspots in PageRank( ) using Profiling.

C. Parallelization of PageRank Algorithm using OpenMP

In the OpenMP implementation, the *generateRandomGraph* function has been parallelized. Specifically, the outer loop of this function, which is responsible for iterating over the vertices of the graph, has been parallelized using *#pragma omp parallel* for static schedule. This parallelization allows multiple threads to execute the loop iterations concurrently, enhancing the performance of the random graph generation process. Additionally, a critical section *#pragma omp critical* is applied within the loop to ensure thread safety during the execution of the *addLink* function, which updates the graph structure. The PageRank computation in the *pageRank* function and memory deallocation in the main function remain sequential to avoid potential race conditions and ensure correct results.

D. Parallelization of PageRank Algorithm using MPI

Message Passing Interface library enables distributed computing across multiple processes. The functions *generateRandomGraph( )* and *pageRank( )* are parallelized. In the *generateRandomGraph( )* function, the loop iterating over vertices is distributed among MPI processes, ensuring each process generates links for a subset of vertices independently. In the *pageRank( )* function, the computation of dangling node contributions and the iterative PageRank updates are distributed among processes, with MPI communication primitives such as *MPI_Allreduce* and *MPI_Allgather*. They are used to synchronize and exchange data between processes. The parallelization allows the algorithm to scale across a cluster of processors, and thereby improving the efficiency of both random graph generation and PageRank computation in a distributed computing environment.

E. Parallelization of PageRank Algorithm using CUDA

In CUDA implementation, the pageRank function is parallelized by offloading the computation of PageRank values to the GPU using a CUDA kernel named updatePageRank. The kernel is executed in parallel by multiple threads. Each thread is responsible for updating the PageRank value of a specific vertex. Shared memory synchronization, such as *__syncthreads( ),* is used to ensure correct calculations within each thread block. The CUDA kernel utilizes atomic operations and shuffle instructions for efficient parallel reduction to calculate the dangling node contributions. This parallelization harnesses the power of the GPU to accelerate the iterative PageRank computation, enhancing the overall performance of the algorithm.

IV. EXPERIMENTAL RESULTS

The PageRank algorithm is implemented using OpenMP, MPI and CUDA and the execution time is measured. Figure 4 illustrates the execution time measured for different numbers of threads using OpenMP implementation. It can be observed that the execution time is minimum when the number of threads is 16 and there is no reduction in execution time when the number of threads is more than 16. The parallelization becomes ineffective irrespective of addition of new threads after 16 threads. MPI implementation of PageRank algorithm is tested with a number of processes as shown in Figure 5. In the MPI implementation, minimum execution time is achieved when the number of processes is 4. The execution time increases after four processes, irrespective of the increase in number of processes. Figure 6 shows the execution time obtained by varying the number of threads per block using CUDA. It can be observed that the execution time is decreased when the blocksize is increased in CUDA implementation. This implementation of PageRank algorithm is executed in Google Colab and the blocksize is limited to 256. Figure 7 shows the comparison of results on parallelizing the PageRank algorithm using OpenMP, MPI and CUDA. The results confirms that our findings are consistent with those in the literature. Specifically:

OpenMP: The performance improvement levels off after 16 threads, similar to what Segel et al. [10] found, where adding more threads leads to extra overhead and doesn't improve efficiency.

MPI: The best performance is achieved with 4 processes, as supported by Manaskasemsak et al. [4], who noted that increasing the number of processes beyond this point adds communication overhead, reducing scalability.

CUDA: The excellent performance of CUDA in our study matches the findings of Kumar et al. [2] and Duong et al. [11], who also found that GPU acceleration significantly benefits graph processing algorithms. This comparison confirms our results and underscores the efficiency of CUDA for large-scale PageRank computations.

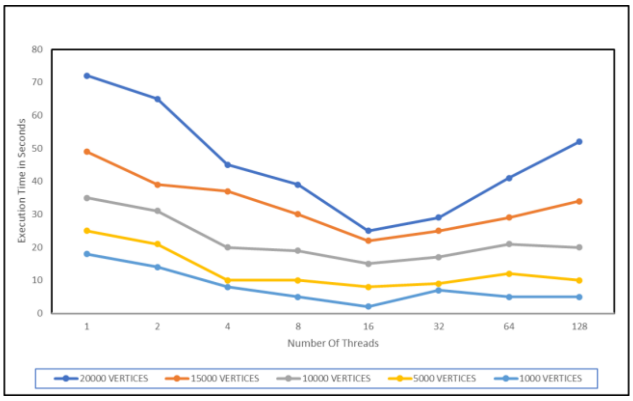


Fig. 4 Execution Time for Varying Number of Threads using OpenMP.

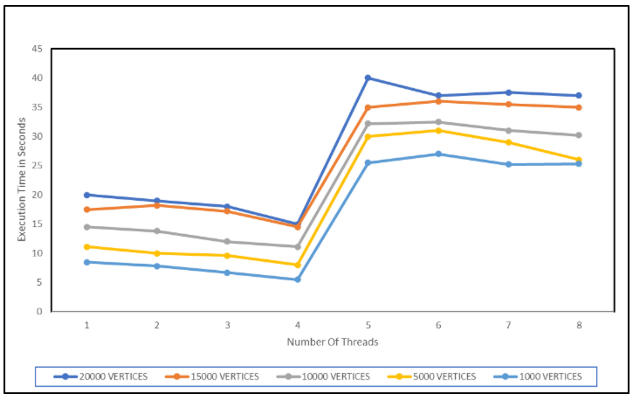


Fig. 5 Execution Time for Varying Number of Threads using MPI.

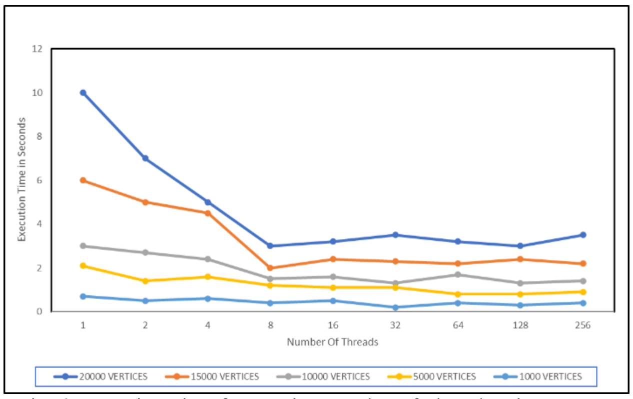


Fig. 6 Execution Time for Varying Number of Threads using CUDA.

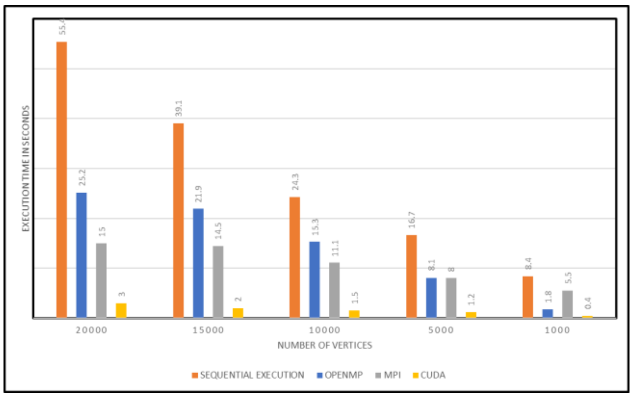


Fig. 7 Comparison of OpenMP, MPI and CUDA Implementations.

V. CONCLUSIONS

The experimental results of parallelizing the PageRank algorithm using OpenMP, MPI, and CUDA are systematically analyzed, and the performance of each parallelization technique is evaluated. The execution times for varying numbers of threads (OpenMP), processes (MPI), and threads per block (CUDA) are measured to understand the scalability and efficiency of each approach. The study demonstrated that CUDA outperformed OpenMP and MPI in terms of execution time, indicating that the GPU-accelerated parallelization using CUDA provided optimal results for the PageRank algorithm. The CUDA implementation took advantage of the parallel processing capabilities of GPUs, resulting in significant speedup and improved efficiency when compared to traditional CPU-based parallelization methods. Profiling using *gprof* highlighted the *pageRank( )* function as a hotspot, emphasizing the need for optimizing this portion of the algorithm. The experimental results further supported the observation, as the parallelization techniques influenced the execution time of this critical function. The blocksize parameter in CUDA implementation showed a noticeable impact on performance, demonstrating the importance of tuning GPU-specific parameters for optimal results.

In summary, the comprehensive experimental results and analysis presented in this work support the conclusion that CUDA provides the most efficient parallelization approach for the PageRank algorithm among other techniques such as OpenMP and MPI.

REFERENCES

[1] Kohlschütter, Christian, Paul-Alexandru Chirita, and Wolfgang Nejdl. "Efficient Parallel Computation of PageRank." In the 28th European Conference on IR Research, London, UK, pp. 241-252, 2006.
[2] Kumar, Tarun, Parikshit Sondhi, and Ankush Mittal. "Parallelization of PageRank on Multicore Processors." In the 8th International Conference, BICDCIT, Bhubaneswar, India, pp. 129-140, 2012.
[3] Cevahir, Ali, et al. "Site-Based Partitioning and Repartitioning Techniques for Parallel PageRank Computation." In the IEEE Transactions on Parallel and Distributed Systems, pp. 786-802, 2006.
[4] Manaskasemsak, Bundit, Putchong Uthayopas, and Arnon Rungsawang. "A Mixed MPI-Thread Approach for Parallel Page Ranking Computation." In the OTM Confederated International Conferences, CoopIS, DOA, GADA, and ODBASE, pp. 1223-1233, 2006.
[5] Desikan, Prasanna Kumar, et al. "Divide and Conquer Approach for Efficient PageRank Computation." In Proceedings of the 6th International Conference on Web Engineering, pp. 233-240, 2006.
[6] Duhan, Neelam, A. K. Sharma, and Komal Kumar Bhatia. "Page Ranking Algorithms: A Survey." In IEEE International Advance Computing Conference, pp. 1530-1537, 2009.

[7] Anakath, A. S., et al. "Optimization of PageRank Algorithm Using Parallelization Method." In AIP Conference Proceedings, vol. 2755, No. 1, 2023.
[8] Yang, Chao-Tung, Chih-Lin Huang, and Cheng-Fang Lin. "Hybrid CUDA, OpenMP, and MPI Parallel Programming on Multicore GPU Clusters." In Computer Physics Communications, vol. 182, no. 1, pp. 266-269, 2011.
[9] Mohamed, Khaled Salah, and Khaled Salah Mohamed. "Parallel Computing: OpenMP, MPI, and CUDA." In Neuromorphic Computing and Beyond: Parallel, Approximation, Near Memory, and Quantum, pp. 63-93, 2020.
[10] Rastogi, Shubhangi, and Hira Zaheer. "Significance of Parallel Computation Over Serial Computation Using OpenMP, MPI, and CUDA." In Quality, IT and Business Operations: Modeling and Optimization, pp. 359-367, 2018.
[11] Duong, Nhat Tan, et al. "Parallel PageRank Computation Using GPUs." In Proceedings of the 3rd Symposium on Information and Communication Technology, pp. 223-230, 2012.
[12] Zhou, Shijie, et al. "Design and Implementation of Parallel PageRank on Multicore Platforms." In IEEE High Performance Extreme Computing Conference, pp. 1-6, 2017.
[13] Yang, Chao-Tung, et al. "Performance Evaluation of OpenMP and CUDA on Multicore Systems." In International Conference on Algorithms and Architectures for Parallel Processing, pp. 235-244, 2012.
[14] Noaje, Gabriel, Michael Krajecki, and Christophe Jaillet. "MultiGPU Computing Using MPI or OpenMP." In Proceedings of the IEEE 6th International Conference on Intelligent Computer Communication and Processing, pp. 347-354, 2010.
[15] Xia, Kun. "The Implementation of Parallel Computation on CPU and GPU." University of Delaware, 2017.
[16] Graham, Susan L., Peter B. Kessler, and Marshall K. McKusick. "Gprof: A Call Graph Execution Profiler." In ACM Sigplan Notices, vol. 17, no. 6, pp. 120-126, 1982.