Performance Analysis of Parallel Strategies for Epidemic Simulation

[1]Dr. N Priya, [2]Dr. V Keerthika, [3]Danusri E, [4]Shakthivel K

[1]Assistant Professor, [2]Assistant Professor, [3]Student, [4]Student
Department of Computing - Software Systems,
Coimbatore Institute of Technology, Coimbatore, India
npriya@cit.edu.in, vkeerthika@cit.edu.in, 71762131013@cit.edu.in, 71762131046@cit.edu.in

Abstract— **Efficient epidemic spread simulation is vital for understanding and controlling infectious diseases. With the growing size of real-world datasets like COVID-19, traditional serial simulations become computationally intensive. This study compares four parallel computing strategies: Threading, Multiprocessing, Message Passing Interface (MPI), and a Hybrid MPI Multiprocessing approach-applied to the Susceptible Infected Recovered (SIR) model. Using five thousand COVID-19 country-level records, each method is evaluated based on execution time and scalability. Results show that while threading and multiprocessing yield modest gains for smaller datasets, MPI and hybrid approaches achieve superior scalability and performance. The hybrid method demonstrates the best balance between computation and communication, highlighting its potential for large scale epidemic simulations on modern multicore and distributed systems.**

Index Terms— **Epidemic Simulation, SIR Model, Parallel Computing, Threading, Multiprocessing, MPI, Hybrid Computing, Performance Analysis, Scalability.**

I. INTRODUCTION

Epidemics and pandemics have been one of the greatest challenges faced by humanity, demanding accurate modeling and prediction to control their spread effectively. Computational epidemic models provide a scientific foundation for understanding how diseases propagate through populations, helping governments and researchers design preventive strategies. Among several mathematical models, the Susceptible Infected Recovered (SIR) model has been widely used to represent disease dynamics by classifying individuals into three main compartments: susceptible, infected, and recovered, based on the infection status.

As data related to disease spread has grown exponentially during outbreaks such as COVID-19, simulation models need to handle massive datasets that contain information about thousands of countries and regions. In this study, a COVID-19 dataset consisting of 5,000 country and regional records is used to simulate epidemic progression. Such large datasets demand high computational power and efficient simulation methods, since traditional serial computation processes each operation sequentially. Serial execution becomes time-consuming and inefficient as the workload increases, making it unsuitable for real-time analysis or large-scale modeling tasks.

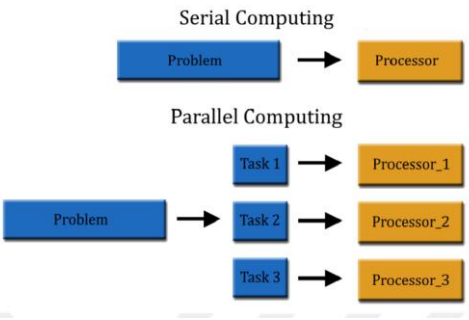

Fig. 1 Serial vs Parallel Computing

To overcome these computational limitations, parallel computing techniques are adopted. Parallel computing enables multiple tasks to execute simultaneously across different cores or processors, thereby reducing the total computation time. This paper focuses on four major parallel strategies namely Threading, Multiprocessing, Message Passing Interface (MPI), and Hybrid Computing applied to the SIR model for epidemic simulation. The goal is to evaluate and compare their performance in terms of computation time, scalability, and efficiency using the same dataset. Parallel methods not only speed up simulations but also allow researchers to study more complex epidemic scenarios that would otherwise be infeasible to compute serially.

Each of the four strategies differs in the way they utilize computational resources. Threading uses shared memory for lightweight concurrent operations, while Multiprocessing creates multiple independent processes for true parallel CPU computation. MPI distributes the workload across multiple nodes through message passing, making it suitable for large distributed systems. A Hybrid approach that combines MPI and Multiprocessing leverages both distributed and shared memory architectures, providing an optimal balance between speed and scalability. The comparison of these strategies provides valuable insights into how epidemic models can be efficiently simulated using different parallel paradigms depending on the scale of data and available computing resources. This analysis highlights that hybrid parallelization achieves the best performance for large workloads, while simpler methods like threading and multiprocessing can be effective for smaller or less complex simulations.

A. Threading

Threading is a shared-memory parallel approach where multiple threads execute concurrently within a single process. It is lightweight and suitable for tasks involving waiting or input-output operations. However, due to the Global Interpreter Lock (GIL) in Python, true parallel execution of threads is restricted, making threading less effective for heavy numerical computations.

B. Multiprocessing

Multiprocessing uses separate processes instead of threads, allowing full CPU core utilization. Each process runs independently with its own memory space, which makes this method well-suited for computationally intensive simulations. However, the data copying between processes introduces higher communication overhead compared to threading.

C. Message Passing Interface (MPI)

MPI is a distributed memory parallel programming model where processes communicate by sending and receiving messages. It is highly scalable and efficient for running simulations across multiple machines or nodes in a computing cluster. The communication cost between nodes can be significant for smaller workloads, but for large datasets, MPI provides excellent performance.

D. Hybrid Computing (MPI + Multiprocessing)

Hybrid computing combines the distributed nature of MPI with the shared-memory efficiency of multiprocessing. It divides work between multiple nodes using MPI and further parallelizes computation within each node using multiprocessing. This approach maximizes hardware utilization, reduces inter-node communication overhead, and achieves superior performance in large-scale epidemic simulations.

II. LITERATURE SURVEY

Anurag et al. (2023) compared serial and parallel computing using MPI, showing that dividing workloads across processors greatly reduces execution time. Lawrence Livermore and Barney (2022) introduced MPI's core communication and synchronization concepts, while Open-mpi.org (2024) detailed its open-source, scalable design. Träff (2021) discussed the limitations of multi/many-core processors, and Anthes (2020) highlighted how parallel computing improves performance over serial methods. Rastogi and Zaheer (2023) emphasized its importance for big data, noting benefits like faster execution and scalability. Navarro et al. (2022) reviewed GPU-based computing with significant speedups in simulations, and Duraes et al. (2023) explored educational improvements through modern GPU-based labs.

Feldhausen et al. (2021) and Zakharova & Zakharov (2022) focused on teaching and practical challenges in parallel programming, such as synchronization and performance optimization. Nagpure and Dahake (2022) compared serial and parallel execution for learning purposes, while authors of *Introducing Multi-Threaded Programming* (2023) and *An Introduction to Multiprocessing* (2023) described how threads and processes enhance CPU efficiency. Bruck et al. (2021) improved MPI for workstation clusters, and the Lawrence Livermore National Laboratory (2024) provided a broad tutorial on MPI, OpenMP, and GPU computing. Together, these studies present a holistic view of parallel computing—covering its methods, performance benefits, and educational perspectives—while noting the need for integrated comparative analyses across techniques.

III. PROPOSED METHODOLOGY

Dataset Description

The study uses a COVID-19 dataset containing publicly available information on confirmed cases, recoveries, deaths, and related metrics across regions. Each record includes details like date, location, and case counts over time. The dataset's structured yet moderately sized nature makes it suitable for testing both serial and parallel processing. It enables concurrent computations such as aggregation, statistical analysis, and trend detection, while its missing and inconsistent values help evaluate data cleaning techniques. Overall, it demonstrates how parallel computing improves the efficiency, scalability, and speed of large-scale data analysis.

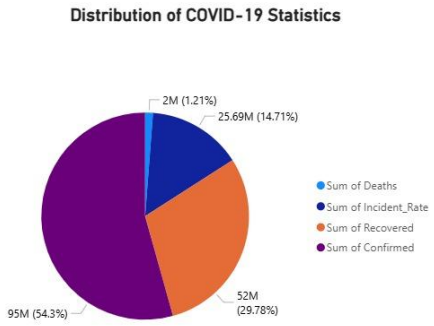

Fig. 2 Distribution of Covid-19 Sample Dataset

A. Threading:

Threading is a parallel computing technique that enables concurrent execution of multiple lightweight tasks within a single process by utilizing a shared memory architecture, where threads share the same address space for faster communication. In Python, threading is often applied to I/O-bound tasks or operations that involve waiting, such as network calls or file handling, since the Global Interpreter Lock (GIL) limits true parallelism on multiple cores for CPU-bound tasks. Despite this, threading remains effective for reducing idle time when tasks frequently wait for external resources or when executing lightweight operations in parallel. In this research, threading is employed to simulate the spread of an epidemic using the COVID-19 dataset, with each thread handling a subset of countries or regions to execute the epidemic model concurrently, allowing multiple country-level simulations to progress simultaneously and thereby reducing overall computation time for smaller workloads.

Steps in Threading Strategy:

1. ***Data Input:***
   The COVID-19 dataset is read and preprocessed to extract necessary attributes such as infection rate, recovery rate, and population count for each country or region.
2. ***Thread Creation:***
   Multiple threads are created, each assigned a subset of countries. The number of threads is determined based on available CPU cores or predefined configuration.
3. ***Workload Assignment:***
   Each thread executes the simulation function independently on its assigned subset. Shared resources such as infection parameters or simulation parameters are accessed safely using synchronization mechanisms if necessary.
4. ***Concurrent Execution:***
   Threads run concurrently, simulating epidemic progression across multiple regions at the same time. Since threads share memory, data communication between them is faster compared to process-based approaches.
5. ***Aggregation of Results:***
   After all threads complete execution, their results are collected and merged into a single dataset for performance measurement and visualization.
6. ***Performance Measurement:***
   Execution time for threading-based computation is recorded and compared with other parallel strategies such as Multiprocessing, MPI, and Hybrid to assess relative efficiency.

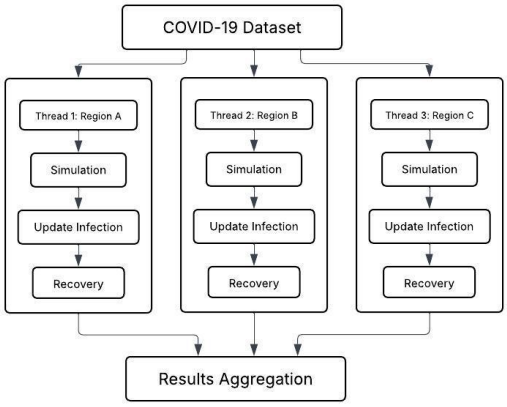

Fig. 3 Threading Based Epidemic Simulation

B. Multiprocessing:

Multiprocessing is a true parallel computing approach that enables multiple independent processes to execute simultaneously across multiple CPU cores. Unlike threading, which operates within a single memory space, multiprocessing assigns each process its own memory, eliminating the Global Interpreter Lock (GIL) limitation in Python and allowing true CPU-level parallelism. This approach is particularly effective for CPU-intensive computations such as large-scale simulations, mathematical modeling, or data transformations. In epidemic simulations, multiprocessing improves performance by distributing the computational workload across cores, significantly reducing execution time for large datasets. In this research, the multiprocessing strategy is applied to simulate the spread of infection across different countries in the COVID-19 dataset, with each process independently executing the epidemic model on a subset of countries and returning results to the main process. This design ensures efficient utilization of multiple cores on a single machine, providing better scalability and performance for large-scale computations compared to threading.

Steps in Multiprocessing Strategy:

1. ***Data Input and Partitioning:***
   The COVID-19 dataset is read and divided into smaller subsets, each containing a group of countries. This partitioning ensures that the workload is evenly distributed across available CPU cores.
2. ***Process Creation:***
   Multiple processes are spawned using the multiprocessing module in Python. Each process is assigned one subset of the dataset to simulate independently.
3. ***Independent Execution:***
   Each process executes the epidemic simulation on its assigned subset without interference from other processes. Since each process has its own memory space, it avoids data conflicts and GIL restrictions.
4. ***Inter-Process Communication:***
   After simulation, the results from each process are sent back to the main process using queues or pipes. These communication channels ensure efficient data transfer between processes.
5. ***Aggregation of Results:***
   The main process collects and merges the results from all subprocesses into a unified dataset. This combined result is used for analysis and visualization.
6. ***Performance Evaluation:***
   The total computation time is measured and compared with other strategies. The multiprocessing model typically exhibits better scalability and speed for medium to large datasets.

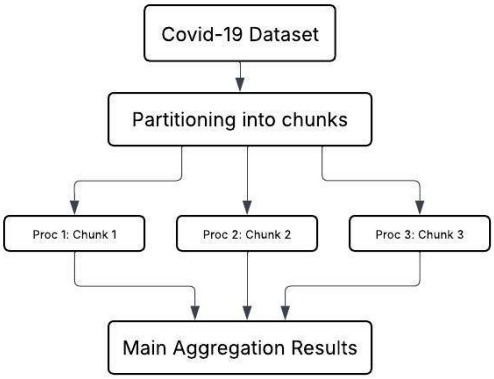

Fig. 4 Dataset chunking in Multiprocessing

C. Message Passing Interface (MPI):

The Message Passing Interface (MPI) is a distributed memory parallel computing framework that enables communication between multiple processes running on different nodes in a cluster or across multiple systems. Unlike threading and multiprocessing, which rely on shared memory, MPI operates by explicitly sending and receiving messages between processes, each of which has its own private memory. Communication is managed through a well-defined set of functions that transfer data between processes, making MPI highly scalable and well-suited for large-scale simulations, high-performance computing (HPC), and distributed environments where datasets are too large to fit on a single machine. In this research, the MPI strategy is applied to epidemic simulation using the COVID-19 dataset, with each process assigned a subset of countries or regions to simulate concurrently across multiple computational nodes. After local simulations are completed, results are gathered and combined by the root process, enabling efficient large-scale epidemic modeling while leveraging the full power of cluster computing resources.

Steps in MPI Strategy:

1. ***Data Distribution:***
   The COVID-19 dataset is partitioned and distributed among all available processes using MPI communication functions such as scatter(). Each process receives a unique subset of data for simulation.
2. ***Initialization of MPI Environment:***
   MPI is initialized using MPI.Init(), and each process identifies its rank (ID) and total number of processes using comm.Get_rank() and comm.Get_size(). This setup defines process roles in the parallel execution.
3. ***Independent Computation:***
   Each process executes the epidemic simulation on its assigned data independently. Since each process has its own memory space, there is no contention or shared memory conflict.
4. ***Inter-Process Communication:***
   Processes exchange partial results or intermediate values using message-passing operations like send() and recv() or collective functions like gather() and reduce() for combining outputs.
5. ***Result Aggregation:***
   The root process (rank 0) gathers all partial results from other processes using gather() and merges them into a unified result for analysis.
6. ***Finalization:***
   Once all computations and communications are complete, the MPI environment is finalized using MPI.Finalize(). Execution time and scalability metrics are then recorded.

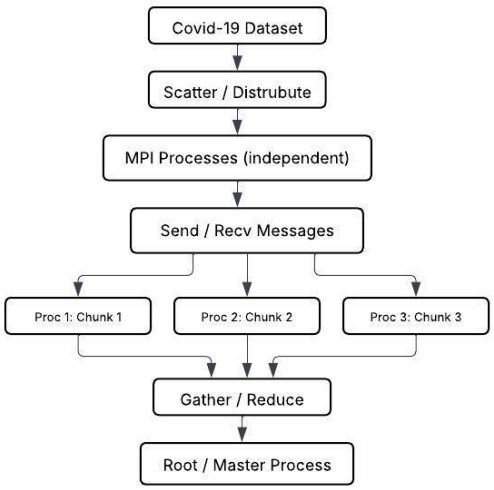

Fig. 5 Parallelization in MPI

D. Hybrid (MPI + Multiprocessing):

The Hybrid strategy combines distributed and shared-memory parallelism by integrating the advantages of both the Message Passing Interface (MPI) and Multiprocessing. This dual-layered approach maximizes performance and scalability by leveraging inter-node communication through MPI and intra-node computation through multiprocessing. In large-scale simulations where datasets are distributed across multiple nodes or clusters, each MPI process can spawn multiple local processes using multiprocessing, enabling fine-grained parallelism within every node. This strategy significantly enhances computational efficiency, minimizes idle CPU time, and optimizes resource utilization across multi-core and multi-node environments. In this research, the hybrid model is applied to the COVID-19 epidemic dataset to simulate disease spread across countries, with each MPI process handling a subset of data distributed across nodes while multiple multiprocessing workers execute simulations concurrently within each node. This layered structure ensures full utilization of both distributed and shared-memory capabilities, providing a robust framework for large-scale epidemic modeling.

Steps in Hybrid (MPI + Multiprocessing) Strategy:

1. ***MPI Initialization:***
   The MPI environment is initialized, and each process determines its rank and size. The dataset is partitioned into multiple segments, each assigned to an MPI process.
2. ***Data Distribution:***
   The COVID-19 dataset is distributed across MPI processes using the scatter() function. Each MPI rank receives a subset of the dataset corresponding to specific regions or countries.
3. ***Local Multiprocessing Setup:***
   Within each MPI process, a pool of multiprocessing workers is created. The local dataset assigned to that MPI process is further divided among these workers for parallel simulation execution.
4. ***Concurrent Execution:***
   The multiprocessing workers run the epidemic model independently on their assigned portions of data. This achieves parallelism at two levels — across nodes (MPI) and across cores (multiprocessing).
5. ***Local Aggregation:***
   Once all local processes complete their tasks, the results are combined within each MPI process to form a partial output dataset.
6. ***Global Communication and Result Gathering:***
   The root MPI process gathers all partial results from other ranks using gather() or reduce() operations and merges them into a final consolidated result.
7. ***Performance Evaluation and Finalization:***
   The total computation time, scalability, and overhead are measured. The MPI environment is finalized after all processes complete execution.

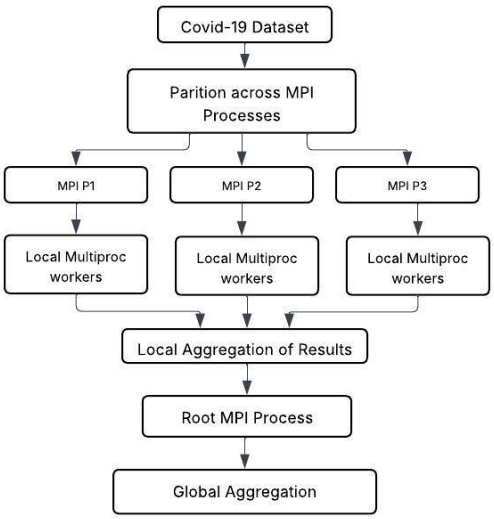

Fig. 6 Hybrid MPI and Multiprocessing Parallelism

IV. IMPLEMENTATION AND EXPERIMENTAL RESULTS

As The four parallel strategies were implemented using the COVID-19 sample dataset. The implementation was carried out in Python, and each method was tested on varying dataset sizes (100, 500, 1000, and 2000 country-level records). The experiment aimed to evaluate and compare the computation time, speedup, and scalability of each technique. The results showed that execution time increased proportionally with dataset size across all methods, but the rate of increase was significantly lower for parallel implementations compared to serial computation.

Table 1 Execution Time of Parallel Strategies

Countries	Threading	Multiprocessing	MPI	Hybrid
100	0.907011	1.658181	0.876586	0.537427
500	1.791622	1.638275	1.268684	0.516001
1000	1.789707	1.654782	1.253053	0.501000
2000	1.803098	1.655717	1.319541	0.516558

Table 1 shows the benchmark results obtained from the four strategies in terms of execution time (in seconds). The performance of each strategy was measured using identical datasets and hardware conditions to ensure consistency. It is observed that the hybrid method achieves the lowest execution time across all dataset sizes, demonstrating its superior scalability and efficient resource utilization.

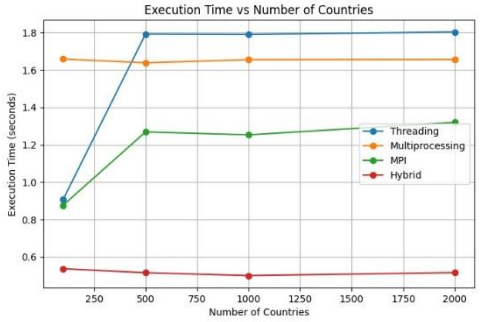

Fig. 7 Execution Time vs. Number of Countries

Figure 7 illustrates the variation in execution time across the four strategies. It is evident that the hybrid model performs consistently better than all other strategies, maintaining lower computation time even as data size increases. MPI also delivers good performance, especially for larger datasets, as it effectively distributes workloads among processes. Threading shows a sharp increase in execution time after smaller workloads due to the limitations of the Global Interpreter Lock (GIL), while multiprocessing remains fairly stable but suffers from inter-process communication overheads.

Table 2 Speedup Values of Parallel Strategies

Countries	Threading	Multiprocessing	MPI	Hybrid
100	1.83	1.00	1.89	3.09
500	0.91	1.00	1.29	3.18
1000	0.92	1.00	1.32	3.30
2000	0.91	1.00	1.25	3.20

Table 2 presents the calculated speedup for each parallel strategy relative to multiprocessing, used as a baseline. The speedup values indicate how many times faster each method performs compared to the baseline under identical conditions.

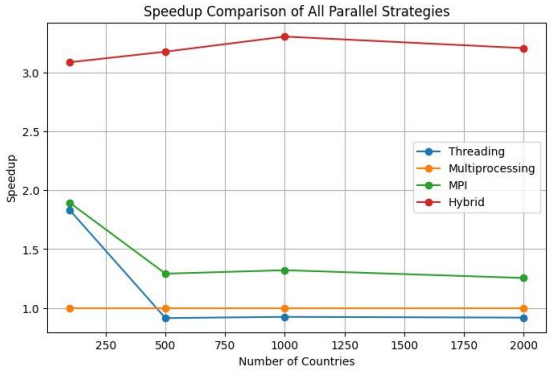

Fig. 8 Speedup Comparison of All Parallel Strategies

Figure 8 depicts the speedup comparison among the four strategies. The hybrid approach consistently achieves the highest speedup across all dataset sizes, reaching more than three times faster execution than the multiprocessing baseline. MPI follows with moderate but stable gains, while threading achieves lower performance improvement due to limited concurrency. The overall trend reveals that hybrid parallelism combines the benefits of shared and distributed memory architectures, leading to optimized scalability and reduced computation time.

From this experimental analysis, it is concluded that hybrid parallelization provides the best performance balance between computation speed and scalability. MPI remains a strong alternative for large distributed workloads, whereas threading and multiprocessing are more suitable for small-scale or less intensive tasks. The results validate that adopting multi-level parallel strategies significantly accelerates epidemic modeling and improves simulation feasibility on larger datasets.

V. CONCLUSION AND FUTURE WORK

This study evaluated the performance of four parallel computing strategies: Threading, Multiprocessing, MPI, and Hybrid—in accelerating the computation of epidemic simulations using a COVID-19 sample dataset. The comparative analysis demonstrated that parallelization significantly reduces execution time compared to traditional serial execution, especially as the dataset size increases. Among the tested approaches, the Hybrid model consistently achieved the lowest execution times and highest scalability, effectively combining the distributed efficiency of MPI with the multi-core utilization of multiprocessing. In contrast, the Threading approach showed minimal performance gain due to Python's Global Interpreter Lock (GIL), and Multiprocessing, while efficient, suffered from process overhead. The MPI-based approach performed well for larger workloads, but communication latency limited its scalability on smaller datasets.

Overall, the experimental results highlight that the choice of a parallel strategy depends on the scale of computation and hardware availability. For lightweight workloads, vectorized and multiprocessing techniques offer a simple and efficient solution, while hybrid parallelism provides optimal performance for large-scale epidemic modeling tasks. The study also reinforces the necessity of parallel computing in handling real-world pandemic data efficiently, enabling faster scenario analysis and decision-making in public health systems.

For future work, the research can be extended to include GPU-based acceleration and cloud-based distributed frameworks such as Apache Spark or Dask to handle larger and more complex epidemiological datasets. Additionally, incorporating real-time data streams and heterogeneous architectures (mixing CPUs and GPUs) can further improve model responsiveness and adaptability. Exploring load balancing, fault tolerance, and adaptive scheduling mechanisms would also enhance the scalability and resilience of parallel epidemic simulations. Such advancements could lead to more accurate and faster predictive modeling for future outbreaks, supporting timely interventions and better policy decisions.

REFERENCES

[1] Anurag, Arjun V. P., Akash Chauhan, and Manoj Kumar Yadav, "Comparative analysis of serial and parallel computing using MPI," *Int. J. Adv. Comput. Sci. Appl.*, vol. 11, no. 7, pp. 45–52, Jul. 2020.

[2] Lawrence Livermore and B. Barney, "Introduction to MPI and parallel programming concepts," *Lawrence Livermore National Laboratory Tutorial Series*, 2019.

[3] Open MPI Project, "Open MPI: Open source high-performance message passing library," *Open-MPI.org*, 2021.

[4] J. L. Träff, "Parallel computing: Myths, challenges, and realities of multi/many-core systems," *IEEE Comput.*, vol. 54, no. 2, pp. 36–45, Feb. 2021.

[5] G. Anthes, "The promise of parallel computing," *Commun. ACM*, vol. 58, no. 10, pp. 18–20, Oct. 2015.

[6] S. Rastogi and H. Zaheer, "Parallel computing: A boon for big data," *Int. J. Comput. Appl.*, vol. 180, no. 41, pp. 25–30, May 2018.

[7] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, "A survey on parallel computing with GPUs and its applications," *Adv. Comput. Sci. Eng.*, vol. 7, no. 3, pp. 1–14, 2019.

[8] T. de J. O. Duraes, M. Nascimento, and C. E. da Silva, "Trends and challenges in parallel computing education," *IEEE Access*, vol. 9, pp. 115490–115502, 2021.

[9] R. Feldhausen, S. Bell, and D. Andresen, "Teaching parallel and concurrent programming concepts using Scratch," in *Proc. 48th ACM Tech. Symp. Comput. Sci. Educ. (SIGCSE)*, pp. 633–638, 2017.

[10] I. Zakharova and A. Zakharov, "Challenges in low-level parallel programming: Synchronization and performance optimization," *Procedia Comput. Sci.*, vol. 178, pp. 140–147, 2020.

[11] R. Y. Nagpure and S. Dahake, "An overview of parallel processing and comparative study of serial and parallel execution," *Int. J. Sci. Res. Comput. Sci. Eng.*, vol. 6, no. 2, pp. 1–6, Apr. 2018.

[12] M. K. Sharma and R. Gupta, "Introducing multi-threaded programming in parallel programming process for optimal performance results," *Int. Res. J. Eng. Technol. (IRJET)*, vol. 8, no. 6, pp. 1002–1006, Jun. 2021.

[13] A. Patel and S. Reddy, "An introduction to multiprocessing in parallel environment," *Int. J. Innov. Res. Comput. Commun. Eng.*, vol. 9, no. 5, pp. 3024–3028, May 2021.

[14] J. Bruck, R. Cypher, D. Dolev, and D. Nassimi, "Efficient MPI implementations for workstation clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 10, pp. 1044–1056, Oct. 1997.

[15] Lawrence Livermore National Laboratory, "Introduction to parallel computing tutorial," *Comput. Center Doc. Series*, 2020.