ISSN: 2584-0495 Vol. 2, Issue 9, pp. 1197-1202

International Journal of Microsystems and IoT

ISSN: (Online) Journal homepage: https://www.ijmit.org

Parallelization of Scientific Applications with MPI

Sanjay Raju, Astha Paditar, Vikas Gaur and Biswajit Bhowmik

Cite as: Raju, S., Paditar, A., Gaur, V., & Bhowmik, B. (2024). Parallelization of Scientific Applications with MPI. International Journal of Microsystems and IoT, 2(9), 1197–1202. https://doi.org/10.5281/zenodo.14100106

Published online: 23 Sept 2024

DOI: https://doi.org/10.5281/zenodo.14100106

International Journal of Microsystems and IoT
Vol. 2, Issue 9, pp.1197-1202; DOI: https://doi.org/10.5281/zenodo.14100106

Parallelization of Scientific Applications with MPI

Sanjay Raju, Astha Paditar, Vikas Gaur and Biswajit Bhowmik

Maharshi Patanjali CPS Lab, BRICS Lab, Department of Computer Science and Engineering, National Institute of Technology Karnataka, Mangalore, Bharat

ABSTRACT
Modernizing scientific codes to harness parallel computing is essential for significantly boosting performance and efficiency. However, transitioning from sequential to parallel programming introduces complex challenges, such as managing global variables, addressing aliasing issues, and integrating random number generators and stateful functions. To address these challenges, this paper proposes a semi-automatic methodology designed to simplify the parallelization of applications with minimal redesign effort. This versatile approach supports various parallel computing paradigms, including shared memory systems (via OpenMP), message passing (via MPI), and GPU computing (via OpenACC). The methodology's efficacy is validated by applying it to four real-world physics and materials science codes, demonstrating its broad applicability and substantial impact on advancing scientific computations.

KEYWORDS
Open Multi-Processing(OpenMP); Message Passing Interface(MPI); Open Accelerators(OpenACC); Scalable Local Fourier Analysis (SLFA); Block Diagonal Varying System (BDVS).

1. INTRODUCTION

High-performance computing (HPC) is characterized by its ability to perform complex calculations at unprecedented speeds. For instance, a typical laptop with a 3 GHz processor can execute around 3 billion calculations per second—an impressive feat compared to human capability but modest relative to HPC systems that can achieve quadrillions of calculations per second. A prime example of HPC is the supercomputer, which consists of thousands of interconnected compute nodes working in parallel to tackle complex tasks [1]. This parallel processing model is akin to aggregating the computing power of thousands of PCs to accelerate task completion significantly.

The shift toward Parallel computing architecture, such as OpenMP, OpenACC, and MPI, are pivotal in advancing the efficiency of scientific codes within the HPC domain. These models enable the parallelization of computational tasks, resulting in substantial performance gains for scientific applications [2]. By distributing computations across multiple processors, parallel computing not only reduces execution times but also handles larger datasets more effectively [3,4]. This process involves converting sequential code into a parallel format, addressing challenges such as loop parallelism and dependency management [5].

The process of transitioning from sequential to parallel code is fraught with challenges. One of the primary difficulties lies in handling data dependencies and ensuring that parallel tasks do not interfere with each other [6,7].

Additionally, selecting the most appropriate parallelization model for a given application can be complex, requiring careful consideration of the specifics and requirements of the code.

Achieving scalability, where performance improvements continue as more processors are added, is another significant challenge [8,9]. Moreover, the necessity for domain-specific knowledge often complicates the parallelization process, as understanding the scientific context is crucial for effective optimization [10,11].

The method of parallelizing scientific applications using OpenMP, OpenACC, and MPI is mentioned in [12]. This approach emphasizes the efficient modernization of sequential codes to make them parallel-ready with minimal redesign. It involves a structured framework that identifies parallelizable loops, evaluates performance gains, converts loops into independent processes, and integrates checkpointing logic. This method effectively addresses the intricacies of loop parallelism and provides strategies to optimize code for enhanced parallelism while minimizing extensive redesign efforts.

The implementation of this methodology led to significant speedups in various scientific codes. Notably, there is a twelvefold increase in the performance of the DiskMass Survey code and a 3.70x enhancement in the efficiency of the Spray Web code. These improvements are achieved through the utilization of parallelization models such as MPI, OpenACC, and OpenMP. However, the study also acknowledges certain limitations. It does not offer explicit solutions for handling true dependencies, choosing suitable parallelization models, achieving scalability, or addressing domain-specific knowledge requirements—challenges that remain in the parallelization of

scientific codes. Despite these limitations, the paper demonstrates notable achievements in enhancing the performance of scientific applications.

The rest of this paper is organized as follows: Section 2 provides a related works. Section 3 presents a proposed methodology. Section 4 discusses the experimental setup and results. Section 5 concludes the final paper.

2. RELATED WORKS

HPC has seen significant advancements in recent years, driven by the need to process vast amounts of data and perform complex calculations more efficiently. Numerous studies have explored various methodologies for parallelizing scientific applications to leverage the full potential of modern HPC systems. Recent breakthroughs in this domain include optimizing nested loop parallelism through meta-heuristics, enhancing distributed stack programming for parallel processing, and employing deep learning approaches to accelerate the analysis of whole-slide images in virtual pathology.

Mahjoub [13] introduced an advanced technique to enhance parallelism in nested loops by transforming non-uniform loop structures into uniform ones using the Shuffled Frog-Leaping Algorithm (SFLA). Their approach identifies optimal Basis Dependence Vector Sets (BDVS), reducing the size of the Dependence Cone (DCS) and improving loop uniformity. However, challenges such as prolonged execution times and sensitivity to parameter settings remain. Drocco [14] explore a distributed stack for programming with distributed algorithms, achieving linear speedup for single-range algorithms and optimizations for multi-range algorithms. However, the complexity of the performance model for multi-range algorithms presents limitation [15] addresses the analysis of gigapixel-scale whole-slide images (WSIs) in virtual pathology using deep learning models. Their HPC-based approach significantly reduces processing times, demonstrating notable efficiency improvements.

Ying [16] presents a technique to overcome speculative parallelization obstacles using the T4 compiler, addressing issues like incorrect assumptions and hardware scalability constraints. While promising, further work is needed to simplify implementation and reduce programming load. The author in [17] employed an optimizing parallel communication by leveraging MPI-4.0 technologies to enhance thread-to-thread communication and eliminate bottlenecks. Their research shows performance improvements and better scalability in MPI+threads applications. Gupta [18] tackle cloud computing challenges by integrating a primary backup model to enhance fault tolerance and productivity. Their E-DFT design algorithm effectively reduces processing time and improves practical features [19] proposed a method to model performance and predict speedup of parallel loops in multi-socket multi-core architectures using M/M/1/N/N queueing models. Their LoopPerf library offers fine-grained control over worker threads, optimizing parallel loop performance. Giordano [20] introduce a method to balance performance between nodes in a cellular automata (CA) model, achieving significant latency reductions in both shared and distributed memory configurations. In cloud computing, venture scheduling [11, 12] aims to allocate tasks to ensure maximum profit and timely completion, despite the challenges posed by dynamic assignment arrivals and limited capacity. The Profit Maximization Algorithm (PMA) [13] employs simulated annealing particle swarm optimization (SAPSO) to optimize project allocation and balance workloads between private and public clouds, addressing the complexities of dynamic scheduling. This paper addresses the limitations identified in these studies by providing a semi-automatic methodology for parallelizing scientific applications. Table 1 summarized the literature.

Table. 1 Existing Approach for Parallelization Scientific Applications

Author	Approach	Results	Limitations
Aldinucci et al. [12]	Parallelize scientific applications using MPI	Speedup of 12 in scientific codes	Unable to achieve scalability
Mahjoub et al. [13]	Convert nested loops into loop parallelization with Shuffled frog-leaping algorithm	BDVS sets discovered by uniformization	Sometimes execution time is large
Drocco et al. [14]	implementation of stack for distributed algorithms	Two Intel Xeon with parallel execution	Sometimes not accurately reflect real-world situations
Li et al. [15]	Parallelization scientific algorithm focusing on digital pathology	Stride size affects the sensitivity of lesion-level detection performance.	Challenging to measure sustained memory bandwidth
Ying et al. [16,24]	Increases the prevalence of parallelism in programming	Creates tiny tasks quickly and removes stack contention	Complexity is high, and resource utilization causes overhead
Zambre et al. [17]	Optimizes thread-to-thread communication independence	Gain in MPI+ thread applications performance	Synchronization issues in partitioned communication models
Gupta et al. [18]	Improves resource utilization and minimizes task response time	Effectively minimizes response time and maximizes resource utilization	Network and interconnect technologies are limiting factors
Cho et al. [19]	Models performance and forecasts speedup of parallel loops in multi-socket multicore architectures	Accurately forecasts acceleration	Does not sufficiently handle synchronization and scheduling overhead
Giordano et al. [20,23]	Dynamic techniques to achieve optimal resource utilization	150\% performance gain with complex transition functions.	Relies on mathematical formalizations.

3. PROPOSED METHODOLOGY

This section introduces the proposed methodology,

emphasizing its novel contributions and improvements over existing approaches. Previous work using MPI-based parallelization provided essential insights for our approach [12]. Techniques for enhancing parallelism in nested loops informed our task resizing strategy, while distributed programming in C highlighted the importance of robust execution in distributed systems. Additionally, deep learning models leveraging high-performance computing influenced our focus on task size variations [15]. Insights into speculative parallelization and thread communication guided our proposed scheduler optimization. Furthermore, dynamic scheduling and performance modeling emphasized task length and scheduling effectiveness, while dynamic load balancing aligned with our goal of optimizing task performance [20,22,26].

Algorithm 1 Task Scheduling Algorithm

```
Initialize PriorityQueue PQ
Define TaskObject with attributes: TaskID, Size, Priority, ArrivalTime
for each task in the list of tasks do
    Create TaskObject with task data
    Add TaskObject to PQ based on Priority
    Use ArrivalTime as a tiebreaker if priorities are the same
end for
Initialize ScheduledTasksQueue
while PQ is not empty do
    Dequeue task from PQ
    if multiple tasks have the same priority then
        Choose the one with the earliest ArrivalTime
    end if
    Check if resources are sufficient for the task
    if resources are available then
        Update available resources
        Add a task to ScheduledTasksQueue for execution
    else
        Re-add task to PQ with updated priority
    end if
end while
Generate a list of tasks with their execution orders from ScheduledTasksQueue
End
```

Fig. 1 Task Scheduling Algorithm

The proposed task scheduling algorithm (Figure 1) initiates a priority queue (PQ) to manage tasks based on their priority. Each task is encapsulated within a Task Object, which contains attributes such as Task ID, Size, Priority, and Arrival Time, which uniquely identify each task, determine resource requirements, and resolve conflicts. The rules iterate through listing obligations, creating a Task Object for everyone, and including it in the PQ based on its precedence. If multiple obligations have equal precedence, Arrival Time is used as a tiebreaker to prioritize the earlier task. After initializing the PQ, the rules create a Scheduled Tasks Queue to maintain the ready tasks for execution.

In the main scheduling loop, tasks are dequeued from the PQ one by one. For each task, the algorithm checks if the necessary resources are available. If sufficient resources are available, they are allocated, and the task is added to the ScheduledTasksQueue for execution. If resources are insufficient, the task is re-added to the PQ with an updated priority to be reconsidered in the next cycle. This process continues until all tasks are scheduled. Finally, the schedule consists of tasks with their execution order derived from the ScheduledTasksQueue, completing the scheduling system. This approach ensures tasks are executed based on priority and arrival time while considering resource constraints.

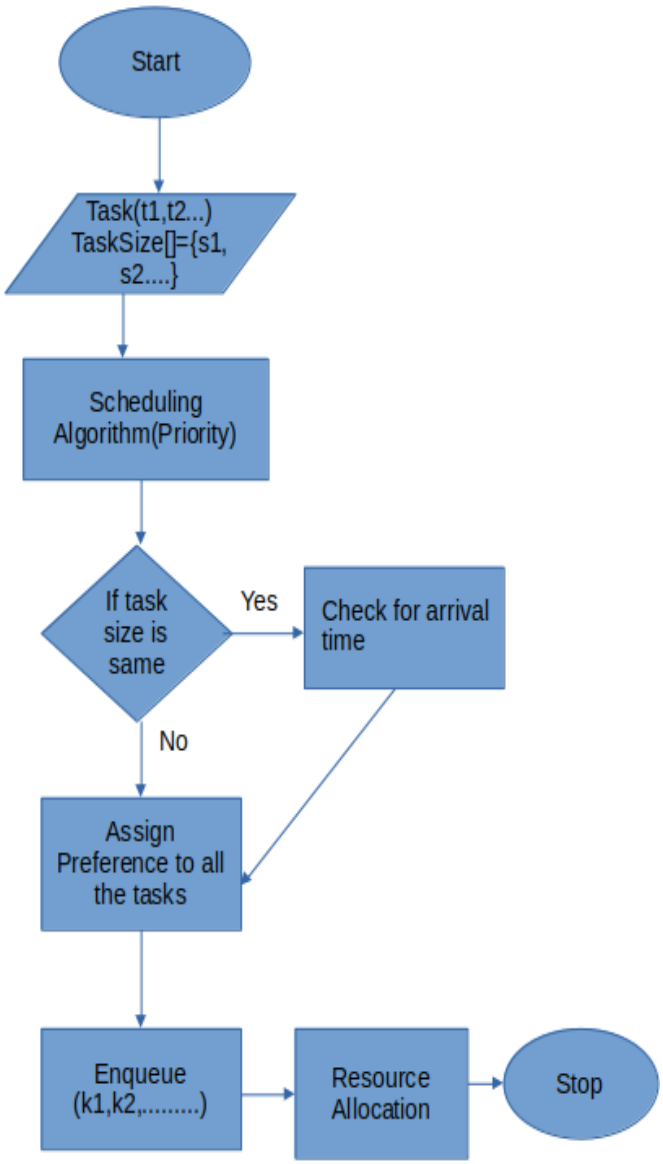

Fig. 2 Proposed Algorithm Flowchart

The proposed approach focuses on dynamic task resizing, optimizing task execution across multiple cores. It enhances scheduler performance by optimizing task sizes and prioritizing tasks based on priority and arrival time. The method incorporates MPI programming for parallel processing, boosting system speed and performance in high-performance computing environments. It ensures effective resource

allocation and reallocation, addressing resource availability and task requirements. The proposed method is designed to scale efficiently with increasing tasks and cores, making it suitable for large-scale scientific applications. This methodology addresses the limitations of previous approaches by providing a semi-automatic system for parallelizing scientific applications, ensuring optimal performance and resource utilization, as illustrated in Figure 2.

4. EXPERIMENT AND RESULTS

This section details the experimental setup, performance metrics, results, and analysis of the proposed task scheduling algorithm. It details the methodology used to evaluate the performance of the scheduling system and includes the results, in Table 2 which highlight the metrics used for performance evaluation. Graphical representations are provided in Figures 3 and 4 to visualize the results effectively. Additionally, the equations used for parallel computing of subtasks and their calculations are presented to support the analysis.

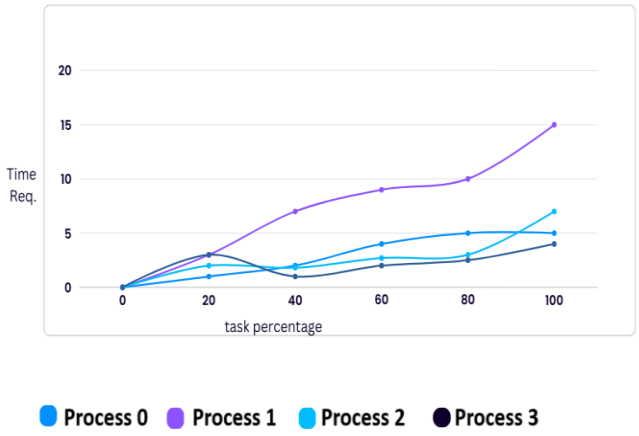

Fig. 3 Parallel Computing Task for Execution Time

4.1 Metrics Used for Performance Measure

A wide range of metrics are calculated and examined to identify grain size and related overheads, which can be dynamically adjusted to optimize task size. Here, Execution Time is (t_{exe}) and Total time is (t_{func}). The key metrics include:

4.1.1 Thread Idle-Rate

This refers to a thread's time in an idle state. A high idle rate suggests the thread frequently anticipates responsibilities or sources in equation 1, which could signify inefficiencies in task scheduling or resource allocation.

$$t_o = \frac{\sum t_{func} - \sum t_{exe}}{n_t} \qquad (1)$$

4.1.2 Task Duration

This is the total time required to finish a task from start to finish. It consists of the time spent actively processing the task and any delays or waiting intervals that could arise during execution. Task in in equation 2. Duration is critical for understanding how long each project will take and scheduling tasks efficiently.

$$t_d = \frac{\sum t_{exe}}{n_t} \qquad (2)$$

4.1.3 Task Overhead

This represents the additional time and resources required to manage a task beyond the actual processing time. It includes various factors like context switching and setup. Reducing task overhead is crucial for enhancing overall system performance mentioned in in equation 3.

$$t_o = \frac{\sum t_{func} - \sum t_{exe}}{n_t} \qquad (3)$$

Table. 2 Evaluation of Parallel Computing Subtasks

Process	Subtask	Task	Result
0	0	0	637
1	1	1	501
2	2	2	474
3	3	3	449
0	4	4	470
1	5	5	648
2	6	6	327
3	7	7	518
0	8	8	628
1	9	9	532
2	10	10	553
3	11	11	533
0	12	12	477
1	13	13	556
2	14	14	343
3	15	15	444
0	16	16	541
1	17	17	502
2	18	18	478
3	19	19	351

4.1.4 Thread Management Overhead

This is the overhead associated with managing threads, including tasks such as creating, destroying, and synchronizing threads. It also covers the cost of context switching between threads. High thread management overhead can affect system performance, so optimizing thread control is essential for efficient multitasking in equation 4

$$T_o = \frac{t_o * n_t}{n_c} \qquad (4)$$

4.1.5 Wait Time

This is the time a thread or assignment spends looking ahead to sources or other conditions to be met before it can hold with execution. Wait time can result from different factors, such as

looking ahead to I/O operations to finish or looking in advance to exceptional tasks or threads to launch assets in equation 5.

$$T_w = \frac{t_d - t_{d1} * n_t}{n_c} \qquad (5)$$

4.1.6 Execution Time (t_{exe})

Execution time is the duration spent actively processing a task, including any extra delays or setup time. It measures the time from when the computation starts to when it finishes, excluding any extra delays or setup time.

4.1.7 Total Time (t_{func})

It represents the total duration of a task, including execution time, delays, overheads, and other factors such as resource waiting, setup, and interruptions. It provides a comprehensive view of the duration of a task, considering all factors affecting its completion.

4.2 Technologies Used

The methodology utilizes various tools and frameworks for performance assessment and optimization. Profilers such as gprof and Valgrind are employed to analyze execution characteristics. Gprof aids in identifying performance bottlenecks by providing detailed records of function calls and execution times. Valgrind, with tools like Callgrind and Cachegrind, offers in-depth analysis of execution time, thread idle costs, and task overhead, which are crucial for performance optimization.

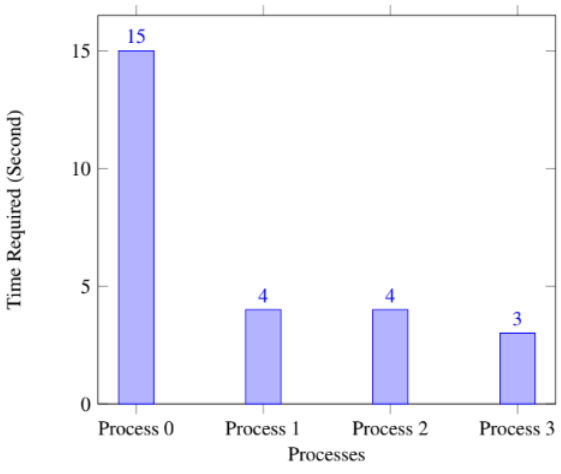

Fig. 3 Time Required for Different Processes

In the realm of parallel computing, OpenMP and MPI are pivotal. OpenMP facilitates task division among multiple threads within a shared memory environment, maximizing the use of multicore processors. MPI, on the other hand, manages communication between processes on different nodes, supporting data exchange and synchronization in distributed computing systems. These frameworks enable efficient parallelization and effective task management.

For visualization purpose, Matplotlibrar and Gnuplot are used to generate graphs and charts that illustrate performance data. SLURM is utilized for job scheduling and resource allocation, ensuring efficient use of computational resources. Additionally, Apache Mesos and Kubernetes are leveraged for dynamic task resizing and load balancing, ensuring tasks are scheduled effectively and resources are optimally utilized. These tools collectively enhance performance and efficiency.

5 CONCLUSION

This paper evaluates the effects of task size variations on the performance of schedulers in multicore systems. The findings reveal that significant performance differences arise with changes in task size, underscoring the importance of task granularity for optimizing scheduling efficiency. The analysis indicates that tasks with durations of 15, 4, 4, and 3 seconds show varied impacts on scheduler performance, demonstrating the need for precise task size adjustment to enhance both execution speed and overall system efficiency. Looking beforehand, we are considering the possibility of dynamically converting project sizes for ongoing development, analyzing the effectiveness of numerous scheduling approaches with exceptional undertaking sizes to encompass a much wider variety of workloads and realistic applications to corroborate our effects similarly.

REFERENCES

1. Czarnul, P. (2018). Parallel programming for modern high performance computing systems. CRC Press.zzavi B. (2012) Design of Analog CMOS Integrated Circuit. Tata McGraw Hill Education.
2. Saxena, D., & Bhowmik, B. (2024, May). Analysis of Selected Load Balancing Algorithms in Containerized Cloud Environment for Microservices. In *2024 IEEE 4th International Conference on VLSI Systems, Architecture, Technology and Applications (VLSI SATA)* (pp.1-6).IEEE. https://doi.org/10.1109/VLSISATA61709.2024.10560139
3. Kirk, D. B., & Wen-Mei, W. H. (2016). Programming massively parallel processors: a hands-on approach. Morgankaufmann.
4. Patterson, D. (2010). In Praise of Programming Massively Parallel Processors: A Hands-on Approach. Parallel Computing.
5. Hwang, K., & Jotwani, N. (1993). *Advanced computer architecture:parallelism,scalability,programmability* (Vol. 199). New York: McGraw-Hill.
6. Li, X. F. (2016). *Advanced design and implementation of virtualmachines*.CRCPress. https://doi.org/10.1201/9781315386706
7. McCool, M. D. (2008). Scalable programming models for massively multicore processors. *Proceedings of the IEEE, 96*(5), 816-831. 10.1109/JPROC.2008.917731
8. Kumar, V. P., & Gupta, A. (1994). Analyzing scalability of parallel algorithms and architectures. *Journal of parallel and distributed computing*, *22*(3),379-391. https://doi.org/10.1006/jpdc.1994.1099
9. Campanoni, S., Agosta, G., Crespi Reghizzi, S., & Di Biagio, A. (2010). A highly flexible, parallel virtual machine: Design and experience of ILDJIT. *Software: Practice and Experience*, *40*(2), 177-207. http://dx.doi.org/10.1002/spe.950
10. Dimakopoulos, V. V. (2013). Parallel programming models. In *Smart multicore embedded systems* (pp. 3-20). New York, NY: Springer New York.
11. Jin, H., Jespersen, D., Mehrotra, P., Biswas, R., Huang, L., & Chapman, B. (2011). High performance computing using MPI and OpenMP on

multi-core parallel systems. *Parallel Computing*, *37*(9), 562-575.https://doi.org/10.1016/j.parco.2011.02.002
12. Aldinucci, M., Cesare, V., Colonnelli, I., Martinelli, A. R., Mittone, G., Cantalupo, B., ... & Drocco, M. (2021). Practical parallelization of scientific applications with OpenMP, OpenACC and MPI. Journal of parallel and distributedcomputing, 157,13-29.. https://doi.org/10.1016/j.jpdc.2021.05.017
13. Mahjoub, S., Golsorkhtabaramiri, M., Amiri, S. S. S., Hosseinzadeh, M., & Mosavi, A. (2022). A New Combination Method for Improving Parallelism in Two and Three Level Perfect Nested Loops. *IEEE Access*, *10*,74542-74554. http://dx.doi.org/10.1109/ACCESS.2022.3190483
14. Drocco, M., Castellana, V. G., & Minutoli, M. (2020, June). Practicaldistributed programming in C++. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing* (pp.35-39). https://doi.org/10.1145/3369583.3392680
15. Li, W., Mikailov, M., & Chen, W. (2023). Scaling the inference of digital pathology deep learning models using cpu-based high-performance computing. *IEEE Transactions on Artificial Intelligence*, *4*(6),1691-1704. http://dx.doi.org/10.1109/TAI.2023.3246032
16. Ying, V. A., Jeffrey, M. C., & Sanchez, D. (2020, May). T4: Compiling sequential code for effective speculative parallelization in hardware. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture(ISCA)* (pp.159-172).IEEE. https://doi.org/10.1109/ISCA45697.2020.00024
17. Zambre, R., Sahasrabudhe, D., Zhou, H., Berzins, M., Chandramowlishwaran, A., & Balaji, P. (2021). Logically parallel communication for fast mpi+ threads applications. *IEEE Transactions on ParallelandDistributedSystems*, *32*(12),3038-3052. http://dx.doi.org/10.1109/TPDS.2021.3075157
18. Gupta, P., Sahoo, P. K., & Veeravalli, B. (2021). Dynamic fault tolerant scheduling with response time minimization for multiple failures in cloud. *Journal of Parallel and DistributedComputing*, *158*,80-93. https://doi.org/10.1016/j.jpdc.2021.07.019
19. Cho, Y., Oh, S., & Egger, B. (2019). Performance modeling of parallel loops on multi-socket platforms using queueing systems. IEEE Transactions on Parallel and DistributedSystems,31(2),318-331.https://doi.org/10.1109/TPDS.2019.2938172
20. Giordano, A., De Rango, A., Rongo, R., D'Ambrosio, D., & Spataro, W. (2020). Dynamic load balancing in parallel execution of cellular automata. IEEE Transactions on Parallel and Distributed Systems, 32(2), 470-484. https://doi.org/10.1109/TPDS.2020.3025102
21. Girish, K. K., Kumar, S., & Bhowmik, B. R. (2024). Industry 4.0: Design Principles, Challenges, and Applications. *Topics in Artificial Intelligence Applied to Industry4.0*,39-68. https://doi.org/10.1002/9781394216147.ch3
22. Hegde, A., & Bhowmik, B. (2024, April). Big Data Insights: Pioneering Changes in FinTech. In *2024 IEEE 9th International Conference for Convergence in Technology (I2CT)* (pp. 1-6). IEEE. https://doi.org/10.1109/I2CT61223.2024.10543820
23. Vaishnavi, V. G. S. S., & Bhowmik, B. (2024, January). Evolution of Neuromorphic Computing. In *2024 Fourth International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)* (pp. 1-8).IEEE. https://doi.org/10.1109/ICAECT60202.2024.10469389
24. Kumar, S., & Bhowmik, B. (2024, January). Emergence, Evolution, and Applications of Cyber-Physical Systems in Smart Society. In *2024 Fourth International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies(ICAECT)* (pp.1-8).IEEE. https://doi.org/10.1109/ICAECT60202.2024.10468864
25. Samimi, N., Nasri, M., Basten, T., & Geilen, M. (2024, May). Work in Progress: Guaranteeing weakly-hard timing constraints in server-based real-time systems. In 2024 IEEE 30th Real-Time and EmbeddedTechnology and Applications Symposium (RTAS) (pp.402-405).IEEE.562-575. https://doi.org/10.1109/RTAS61025.2024.00043
26. Hammadeh, Z. A., Quinton, S., & Ernst, R. (2019). Weakly-hard real-time guarantees for earliest deadline first scheduling of independent tasks. *ACM Transactions on Embedded Computing Systems(TECS)*, *18*(6),1-25. https://doi.org/10.1145/3356865

AUTHORS

Sanjay Raju is currently pursuing M.Tech in Computer Scienceand Engineering from NIT Karnataka, Mangalore, India. He received the BTech degree in Computer Science and Engineering from Vidya Jyothi Institute of Technology, Hyderabad, India in 2020. His area of interest includes deep learning and computational performance.

E-mail:ramapogusanjayraju.232cs026@nitk.edu.in

Astha Paditar recieved Btech degree from Jabalpur Engineering College, Madhya Pradesh, India in 2023. She is currently pursuing MTech in Computer Science and Engineering from National Institute of Technology Karnataka, Mangalore. Her research interest includes deep learning, machine learning and computational performance.

Author Email: asthapatidar.232cs003@nitk.edu.in

Vikas Gaur recieved Btech degree from Jabalpur Engineering College, Madhya Pradesh, India in 2023. He is currently pursuing MTech in Computer Science and Engineering from National Institute of Technology Karnataka, Mangalore. His research interest includes deep learning, machine learning and computational performance

E-mail:vikasgaur.232cs037@nitk.edu.in

Biswajit Bhowmik is serving as Assistant Professor at NIT Karnataka, Bharat, in Dept. of Computer Science and Engineering. He has more than 12 years of teaching experience. His research interest includes Circuits and System (Design, Testing, Verification), CPS, IoT, and AI technologies. He has published more than 100 research papers, including 3 Best Paper Awards, 1 Young Scientist Award, and a Best Researcher Award. He has setup a research lab called BRICS where at the moment 25 scholars from UG, PG, and PhD levels are actively involved in researches in the above mentioned areas. He is a member of ACM, senior member of IEEE, and associated with various IEEE societies.

E-mail: brb@nitk.edu.in