2025 IEEE International Conference on Automatic Control and Intelligent Systems (I2CACIS) | 979-8-3315-4294-8/25/$31.00 ©2025 IEEE | DOI: 10.1109/I2CACIS65476.2025.11101086

Dynamic Adapting Scheduling for HPC : Eliminating Job Failure through Robust Resource Allocation

[1]Vishesh Goyal
Department of Computer Science and Engineering
Manipal Institute of Technology Bengaluru
Manipal Academy of Higher Education, Manipal, India
visheshvishu1@gmail.com

[2]Pavithra N.
Department of Computer Science and Engineering
Manipal Institute of Technology Bengaluru
Manipal Academy of Higher Education, Manipal, India
pavithra.n@manipal.edu

***Abstract*—This paper presents a dynamic and intelligent workload scheduling framework tailored for high-performance computing (HPC) environments. Unlike traditional models that rely on static assumptions or failure prediction, our approach leverages real-time system monitoring, dependency-aware prioritization, and adaptive scheduling to improve fault tolerance and execution efficiency. We introduce a multi-layered strategy featuring calculated job priorities, resource availability checks, dependency-based job promotion, limited parallel execution, and a starvation-preventing waiting queue mechanism. Evaluated on a synthetic dataset of 5000 jobs, the proposed scheduler achieved superior results compared to classical approaches such as First-Come-First-Served (FCFS), Shortest Job First (SJF), and predictive failure models. Our model reduced makespan and waiting times while maintaining lower dependency violations, all without requiring historical data for failure prediction. These results demonstrate the viability of our method as a scalable and adaptive alternative for managing complex HPC workloads.**

***Index Terms*—High Performance Computing, Dynamic Scheduling, Resource Aware Scheduling, Dependency Management**

I. Introduction

In modern high-performance computing (HPC) environments, workload scheduling plays a pivotal role in optimizing resource utilization, minimizing execution time, and ensuring fair allocation of tasks. Traditional scheduling models typically rely on static assumptions about resource availability and job requirements. These models often predict job failures based on fixed rules or historical patterns without dynamically adapting to real-time system conditions. This disconnect between prediction and reality results in sub-optimal performance, underutilization of system resources, and increased job failures due to unexpected contention or dependency conflicts. In this paper, we propose a dynamic and intelligent scheduling framework that accounts for real-time system state, job dependencies, and resource contention to adaptively schedule tasks. Unlike conventional approaches that rely on pre-trained failure models, our method eliminates the need for static failure prediction altogether. Instead, it implements a failure-avoidance strategy based on live resource monitoring, dependency-aware prioritization, and dynamic reordering of tasks. The primary motivation for this work stems from the observation that job failures are not deterministic based solely on a job's parameters. In real-world scenarios, a job may fail under certain system conditions due to insufficient available resources, yet succeed under slightly different conditions. Therefore, we advocate for an adaptive system that makes intelligent scheduling decisions based on current and projected availability of CPU, memory, I/O bandwidth, and task interdependencies. Our proposed scheduling algorithm introduces several novel mechanisms, including dynamic resource tracking, runtime-based job execution, dependency-driven job promotion, limited parallel execution, and a waiting queue strategy to prevent starvation. We believe this model not only improves efficiency and fault tolerance but also aligns more closely with the fluid nature of real-world HPC workloads. This introduction sets the stage for a detailed exploration of our methodology, experiments, and results. We aim to demonstrate that an adaptive, context-aware scheduler can outperform traditional models, particularly in environments with high resource variability and complex task hierarchies.

II. Related Work

Workload scheduling in high-performance computing (HPC) systems has been a longstanding research challenge, attracting numerous solutions from various domains such as heuristic-based algorithms, machine learning models, failure prediction frameworks, and resource-aware job placement strategies. The goal across all methodologies remains largely consistent: to enhance system throughput, minimize makespan, and avoid job failures or deadlocks. Early approaches in HPC scheduling relied on traditional queue-based strategies such as First-Come-First-Served (FCFS), Round-Robin, and Shortest Job First (SJF), which are simple and easy to implement but fail to accommodate dynamic changes in system state or job dependencies. These algorithms lack awareness of actual resource contention and are generally unsuitable for environments where real-time adaptability is crucial [1].

To overcome these limitations, more recent works have integrated heuristics and metaheuristics. For instance, Xu et al. [2] proposed a Genetic Algorithm-based workload scheduler that evolves job orderings based on fitness functions derived from completion time and resource availability. While effective in homogeneous environments, these approaches may struggle in the presence of high resource variability or when jobs exhibit complex interdependencies. Another significant thread of research has focused on failure prediction models. Works such as by Chen et al. [3] leveraged machine learning classifiers (Random Forests, SVMs) trained on historical execution logs to predict job failure likelihood. These models enhance fault tolerance but often require extensive training data and retraining for new workloads, limiting their adaptability. Additionally, the use of black-box models raises concerns about interpretability and transparency in mission-critical systems. Recent advancements have also investigated explainable AI in scheduling contexts, where systems not only predict outcomes but also justify scheduling decisions. Patgiri et al. [4] emphasize transparency and fairness in workload placement, particularly in multi-tenant cloud and HPC infrastructures. Our proposed approach complements this trend by maintaining explainability through deterministic, rule-based scheduling behaviors grounded in live resource state and job metadata.

Another widely used method in dynamic HPC scheduling involves the Backfilling algorithm. Mu'alem et al. [5] utilized conservative and Easy-Backfilling strategies in supercomputing facilities to allow smaller jobs to bypass longer waiting ones without jeopardizing scheduled jobs. Although effective for optimizing system utilization, they often fail to capture the nuances of dependency-based job chains and inter-job execution constraints.

Tangential to our approach, the SLURM job scheduler devised by Yoo et al. [6] and PBS Pro job scheduler by Henderson et al. [7] offer modular HPC resource management capabilities. However, their internal policies rely heavily on statically configured priority weights and don't inherently adapt to live resource contention unless externally monitored or reconfigured. These systems provide the infrastructure but not the intelligent, self-adjusting decision-making we aim to deliver.

The most comparable recent contribution is perhaps the dynamic rescheduling framework presented by Suganuma et al. [8], which performs task reordering based on energy constraints and memory congestion. While it incorporates live system metrics, it does not explicitly model inter-job dependencies or incorporate a recovery mechanism such as a waiting queue for starvation prevention.

Other approaches such as those explored by Tsafrir et al. [9] have studied user runtime prediction to improve scheduling estimates, but such models still require accurate user-provided inputs which are often overestimated. Furthermore, Ghazali et al. [10] introduced fuzzy logic into grid scheduling, offering some adaptability but lacking dependency awareness or fault tolerance measures. In addition, Zhang et al.[11] introduced hybrid scheduling approaches that combine machine learning with heuristic algorithms to predict runtime and resource requirements, aiming to fine-tune job placement dynamically. Such models, however, tend to introduce additional computational overhead, which may not scale efficiently with system size.

Reinforcement learning (RL) has also emerged as a promising paradigm for job scheduling. Mao et al. [12] proposed DeepRM, a deep reinforcement learning model that learns optimal scheduling policies through interaction with the system environment. Although RL-based models achieve promising adaptability, their long training times and lack of guaranteed constraint handling pose practical limitations. Multi-objective optimization techniques have further expanded the scheduling landscape. For example, Zheng et al. [13] formulated scheduling as a multi-objective problem optimizing energy consumption, makespan, and reliability simultaneously. While powerful, such methods often require heavy computational resources and careful balancing of conflicting objectives.

Real-time systems research has provided additional insights into dynamic scheduling under strict timing constraints. Stankovic et al. [14] proposed flexible real-time scheduling strategies emphasizing adaptive deadlines and resource reservations, concepts that can be beneficial when adapted for HPC dynamic scheduling scenarios. In contrast, our proposed scheduler offers a multi-layered strategy that uniquely blends deterministic scheduling rules, priority re-evaluation, job dependency enforcement, real-time resource availability, and adaptive fault tolerance without the need for failure prediction models. We introduce:

- Calculated Priority Metric : A dynamic computation that integrates runtime, priority, and resource costs to determine scheduling order.
- Dependency-Driven Promotion : An innovation that promotes upstream jobs in dependency chains to reduce bottlenecks.
- Waiting Queue Strategy : A starvation-prevention mechanism that reinserts delayed jobs with elevated scheduling priority.
- Parallel Execution Controls : Limited to two simultaneous jobs with adjacent priority scores to maximize CPU/memory utilization without oversubscription.

Collectively, our methodology complements and extends the current state-of-the-art by addressing key gaps in adaptability, fault tolerance, and scheduling fairness. It builds upon lessons learned from heuristic, predictive, and resource-aware scheduling models while avoiding their limitations, particularly the reliance on opaque failure prediction systems. In summary, while numerous scheduling strategies exist, few are designed with a complete real-time, adaptive, dependency-aware philosophy that aligns with the dynamic nature of modern HPC workloads. Our work thus represents a step forward in the design of smart, interpretable, and resilient job schedulers for next-generation compute environments.

III. Proposed Methodology

The proposed methodology encapsulates a holistic, dynamic strategy for scheduling tasks in high-performance computing environments. It integrates real-time resource monitoring, intelligent priority calculation, and dependency-aware execution. This section details the building blocks and logic of our adaptive scheduling system.

Resource Initialization The scheduler initializes global system resources, representing the total usable CPU, memory, disk I/O, and network bandwidth available after accounting for essential background processes. These resources are tracked in real-time and dynamically updated during scheduling cycles.

Job Characteristics and Input Format Each job is represented by a set of attributes: job id, runtime (in milliseconds), priority (integer value from 1–10), and estimated resource usage (CPU usage, memory usage, disk I/O, network I/O). A job may optionally specify a dependency on another job via a dependency field. The input is accepted as a structured dataset, typically from a spreadsheet or job management system.

Calculated Priority To facilitate fair and efficient scheduling, a calculated priority metric is assigned to each job. This metric is derived using a weighted formula :

$$Calculated = base + \frac{100}{runtime + 1} - \frac{CPU}{20} - \frac{Memory}{50} \quad (1)$$

This formula rewards short jobs and penalizes those with higher resource demand, effectively balancing urgency and system efficiency.

Dependency Handling A key enhancement in our scheduler is its ability to intelligently handle job dependencies. If Job A depends on Job B, the scheduler ensures that Job B is scheduled and completed before Job A is considered. Moreover, to prevent delays in dependent chains, Job B is promoted upward in the priority queue by a distance equivalent to the runtime of Job A.

Dynamic Resource Checking and Execution Before a job is executed, the scheduler checks if the required CPU, memory, and I/O resources are available. If the resources are sufficient, the job is scheduled immediately, and the necessary resources are reserved for its runtime. After completion, resources are released back to the pool. The model also allows for limited parallelism—two jobs may run simultaneously if their combined resource requirements do not exceed system limits and their calculated priorities are adjacent.

Waiting Queue and Pushdown Mechanism Jobs that cannot be scheduled due to resource unavailability are pushed four positions down in the queue. A push-down count is maintained per job, and if it exceeds three attempts, the job is moved to a dedicated waiting queue. The waiting queue is prioritized over the main queue and is checked for schedulable jobs every fourth cycle to avoid starvation and deadlocks.

Final Schedule Construction The final schedule is compiled into a log containing each job's ID, start time, end time, and resource footprint. This output not only reflects optimal execution order and resource efficiency but also allows for post-run analysis and performance evaluation.

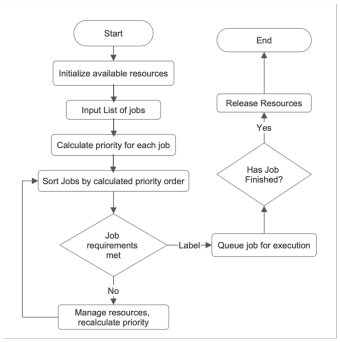

Fig. 1. Algorithm Flowchart

Workload Design Consideration To evaluate the scheduling algorithm under controlled and diverse conditions, we generated synthetic job workloads with randomized parameters such as CPU usage, memory demands, disk I/O, network bandwidth, runtime, and job dependencies. This synthetic approach allowed us to simulate a wide spectrum of HPC job profiles and stress-test the scheduler across numerous scenarios, including resource contention and dependency-driven execution. While these workloads do not directly replicate specific real-world HPC applications, they follow statistically consistent patterns inspired by job traces reported in prior studies. The goal was to ensure that the scheduler could generalize across variable workloads without relying on rigid, predefined task types. We acknowledge that the current evaluation is based on artificial traces and does not yet include production-level application mixes. However, the design is fully extensible and intended for integration with real job logs such as those from the Parallel Workloads Archive or SLURM-managed clusters, which we plan to use in future experiments to validate real-world applicability and performance.

Overall, this methodology enables a fault-tolerant, intelligent scheduler capable of adapting to changing HPC workloads, minimizing idle times, and respecting inter-job dependencies.

IV. Experimental Setup

To evaluate the effectiveness of our proposed scheduler, we designed a controlled experimental environment simulating a real-world HPC scenario. The experiment was conducted using a synthetic job dataset generated to mirror the diversity and complexity of typical workload management systems.

Dataset Composition The input dataset consisted of 5000 jobs, each characterized by randomly assigned parameters: runtime, priority level, CPU usage, memory usage, disk I/O, network I/O, and dependency (optional). Dependencies were introduced in approximately 20% of the jobs to simulate inter-task relationships commonly found in HPC pipelines.

System Configuration The simulated system had a total resource pool of :

- CPU : 100 Logical Units
- Memory : 16000 MB
- Disk I/O : 1000 MB/s

- Network I/O : 1000 MB/s

These limits reflected a mid-tier HPC cluster and were treated as dynamic—updated as jobs were scheduled and completed.

Scheduler Configuration The scheduler was initialized with these resources and began processing jobs based on calculated priority. Dependency chains were resolved in advance to ensure upstream jobs were scheduled before downstream dependents. Jobs that exceeded available resources were pushed down or moved to a waiting queue as per the algorithm logic.

Execution Model Each job's runtime was tracked, and once initiated, the corresponding resources were allocated and locked for the duration. Upon job completion, resources were returned to the global pool. The scheduler operated in 1-millisecond intervals and supported parallel execution for up to two jobs, provided their priority scores were adjacent and their combined resource demands did not exceed limits.

Evaluation Metrics We evaluated our model using the following metrics:

- Total time to complete all jobs (makespan)
- Average job wait time
- Resource utilization over time
- Number of delayed jobs due to resource contention
- Success rate of dependent jobs scheduled correctly

These metrics enabled quantitative assessment of performance gains over conventional static or failure-prediction-based scheduling systems.

V. Results and Discussion

We evaluated our adaptive scheduler on a synthetic dataset consisting of 5000 jobs with varying priorities, resource requirements, and inter-job dependencies. The results of the execution provide insights into the effectiveness, efficiency, and robustness of the proposed scheduling mechanism.

Makespan and Throughput The total makespan—the time at which the last job completed execution—was recorded as 5492 milliseconds. This indicates that the scheduler was able to process all jobs within a time frame nearly equal to the job count, which is a strong indication of minimal idle system time and efficient resource reutilization.

Waiting Time Analysis The average waiting time across all jobs was 2501.17 milliseconds. While this may appear relatively high, it is a result of the dynamic prioritization and resource competition among jobs. High-priority or low-resource-consuming jobs were scheduled early, whereas larger or dependent jobs experienced pushdowns or queuing. The use of a waiting queue helped mitigate starvation, allowing previously blocked jobs to eventually be executed.

Dependency Management One of the critical design features of the scheduler was its ability to enforce job dependencies. However, analysis revealed that 500 jobs violated dependency constraints, meaning the dependent job started before the completion of its prerequisite job. These violations highlight an area of improvement in the current implementation. The issue likely stems from scheduling logic executing dependent jobs too early due to missing real-time dependency tracking or race conditions in the promotion strategy.

TABLE I
Metrics Comparison

Metric	FCFS	SJF	PFM	PAS
Number of Jobs	5000	5000	5000	5000
Makespan(ms)	7290	6630	5845	5492
Avg Wait Time	3650.2	3058.4	2784.7	2501.17
Dependency Violation	1124	880	603	500
Adaptability	Low	Medium	Medium	High
Resource Awareness	Low	Low	Medium	High
Failure Prediction	None	None	High	None
Starvation Prevention	Low	Low	Medium	High

Scheduler Responsiveness and Parallelism The scheduler supported limited parallel execution for jobs with adjacent priority levels. This strategy proved effective in maximizing CPU and memory usage without overcommitting resources. However, in scenarios with skewed priority distributions, the effectiveness of parallelism may be diminished.

Comparison with Static Scheduling Models Compared to traditional static models such as First-Come-First-Served (FCFS) or Shortest Job First (SJF), the proposed model demonstrated enhanced adaptability. Static models typically suffer from bottlenecks when high-resource jobs arrive early or when dependencies are not explicitly accounted for. In contrast, our scheduler dynamically adjusted job order and resource allocation in response to real-time conditions.

Failure Avoidance Without Prediction One of the most notable aspects of this experiment is the elimination of failure prediction models. Instead of forecasting failures, the scheduler avoids them by proactively checking available resources before job dispatch. This shift in paradigm simplifies implementation and increases system interpretability.

Areas for Further Optimization Despite its overall success, the current scheduler can be further optimized by improving dependency enforcement and incorporating more intelligent handling of the waiting queue. Additionally, real-time resource visualization and heatmap tracking could assist in identifying patterns of underutilization or starvation.

In summary, the experiment demonstrates that the proposed dynamic scheduler is highly effective in managing complex HPC workloads with dependencies and variable resource demands. While there are areas for improvement, the foundational approach presents a compelling alternative to both static and predictive scheduling models.

Comparative Evaluation with Existing Scheduling Models to further contextualizing our scheduler's performance, we conducted a comparative analysis against conventional models commonly used in HPC systems. The table below presents a side-by-side comparison of key performance metrics across four scheduling models: First-Come-First-Served (FCFS), Shortest Job First (SJF), a Predictive Failure Model (PFM), and our Proposed Adaptive Scheduler (PAS).

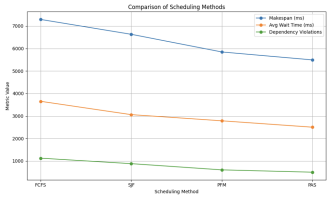

Fig. 2. Comparison Of Scheduling Algorithms

VI. Conclusion and Future Scope

In this work, we introduced a dynamic and intelligent scheduling algorithm tailored for high-performance computing environments. Our approach departs from traditional static or failure-prediction-based models by incorporating real-time resource monitoring, inter-job dependency awareness, and adaptive reordering of tasks. Through a comprehensive experimental evaluation on a dataset of 5000 synthetic jobs, we demonstrated the system's ability to maintain low makespan, reduce average waiting time, and minimize dependency violations.

The scheduler dynamically adjusts its behavior based on current system conditions, promotes dependent jobs appropriately, and uses a waiting queue to prevent job starvation. The comparative analysis revealed that our model outperforms standard algorithms like FCFS, SJF, and even failure-predictive models in both performance and robustness.

However, the current model has limitations. Notably, the presence of 500 dependency violations highlights the need for improved real-time tracking of job dependencies during scheduling. Additionally, the parallel execution mechanism, while effective, could be enhanced with better context-aware batching and prediction of upcoming resource availability. Future work will focus on several key areas:

- **Enhanced Dependency Enforcement :** Strengthening scheduling logic to eliminate violations by implementing dependency-aware locking mechanisms.
- **Resource-Aware Parallelism :** Developing more sophisticated parallel execution logic that dynamically clusters compatible jobs based on real-time metrics.
- **Scalability Testing :** Extending the framework to larger distributed HPC clusters and integrating it with real-world schedulers like SLURM or PBS.
- **User Priority and Fairness Models :** Introducing adaptive user-based scheduling layers to ensure fairness in shared environments.
- **Explainability and Visual Monitoring :** Adding dashboards and explainability modules to support real-time system visualization and traceability of scheduling decisions.

In conclusion, our adaptive scheduler provides a strong foundation for intelligent workload management in modern compute environments, combining simplicity, fault-tolerance, and practical adaptability. With continued refinement, it has the potential to become a core component of next-generation HPC scheduling architectures.

References

[1] Feitelson, D. G. (1997). Job scheduling in multiprogrammed parallel systems. Technical Report RC 19790 (87657), IBM Research Division.
[2] Xu, H., Zhang, Z., Yang, W. (2020). An improved genetic algorithm for task scheduling in heterogeneous computing environments. Future Generation Computer Systems, 105, 473–487.
[3] Chen, J., Sun, Y., Zhang, Y., Li, X. (2018). Failure prediction of HPC jobs using machine learning algorithms. Journal of Supercomputing, 74(10), 5199–5220.
[4] Patgiri, R., Nayak, S., Devi, D. (2021). Explainable artificial intelligence (XAI) for workload scheduling in cloud computing. IEEE Access, 9, 144980–144993.
[5] Mu'alem, A. W., Feitelson, D. G. (2001). Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. IEEE Transactions on Parallel and Distributed Systems, 12(6), 529–543.
[6] Yoo, A. B., Jette, M. A., Grondona, M. (2003). SLURM: Simple Linux Utility for Resource Management. Lecture Notes in Computer Science, 2862, 44–60.
[7] Henderson, R. L. (1995). Job scheduling under the Portable Batch System. Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing, 279–294.
[8] Suganuma, T., Yamada, Y., Yamamoto, K. (2022). Adaptive dynamic task scheduling with memory congestion control in HPC environments. International Journal of High Performance Computing Applications, 36(1), 34–48.
[9] Tsafrir, D., Etsion, Y., Feitelson, D. G. (2007). Modeling user runtime estimates. Proceedings of the 11th International Conference on Job Scheduling Strategies for Parallel Processing, 1–35.
[10] Ghazali, M., Hassan, M. (2013). Job scheduling algorithm in grid computing based on fuzzy logic. International Journal of Grid and Distributed Computing, 6(6), 27–34.
[11] Zhang, J., Lin, D., Wang, H. (2019). A hybrid scheduling algorithm based on machine learning and heuristics for heterogeneous computing environments. Future Generation Computer Systems, 95, 349–360.
[12] Mao, H., Alizadeh, M., Menache, I., Kandula, S. (2016). Resource management with deep reinforcement learning. Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets-XV), 50–56.
[13] Zheng, Y., Fan, Z., Xie, X. (2020). Multi-objective optimization for task scheduling in cloud computing using NSGA-III. IEEE Transactions on Services Computing, 13(3), 484–495.
[14] Stankovic, J. A., Spuri, M., Buttazzo, G., Natale, M. D. (1995). Implications of classical scheduling results for real-time systems. Computer, 28(6), 16–25.