Opportunity-aware Scheduling for Data Operations in Long Running Applications*

Minh-Tri Nguyen, Anh-Dung Nguyen, Jarno Rantaharju, Hong-Linh Truong
Department of Computer Science, *Aalto University*
{tri.m.nguyen, anh-dung.nguyen, jarno.rantaharju, linh.truong}@aalto.fi

***Abstract*—Many data-intensive applications on HPC systems have not fully utilized allocated resources due to inefficient workflow with limited task parallelization. Additionally, storage constraints introduce substantial delays between tasks for executing data-intensive operations in long-running applications. This report presents our design and implementation of the Scheduler in the ODOP framework, which leverages underutilized resources to run opportunistic tasks handling these data-intensive operations. With customizable scheduling algorithms, ODOP scheduler allows non-HPC experts like domain scientists to reduce time-to-solution and improve resource utilization, saving significant operational costs in HPC.**

***Index Terms*—data operations, opportunistic tasks, performance optimization, computational applications**

I. Introduction

A. Background

Many data-intensive applications are heavily relying on high-performance computing (HPC) systems with many-core CPUs and accelerators, producing/handling massive data. Such applications demand substantial computational power, up to thousands of compute nodes over extended periods. Given massive allocations, efficiently utilizing all the resources remains a significant challenge, particularly for domain scientists lacking expertise in HPC and underlying systems. In the previous work, we have introduced ODOP framework, which considers opportunistic tasks (*optasks*) as tasks encapsulating data operations, which can be executed concurrently (time overlapping) with the main application within the allocated resources [1]. For such tasks, opportunistic scheduling (*Op. scheduling*) involves dynamically assigning *optasks* to idle resources, optimizing application performance, and maximizing resource utilization.

This paper presents the design and implementation of the *Op. scheduling* within our ODOP scheduler [1]. *Op. scheduling* is especially beneficial in modern HPC systems with multiple types of resources for intensive computations. For instance, while accelerators (e.g., GPUs) handle heavy computational tasks like deep learning or simulations, CPUs can be used for other tasks such as data preprocessing, logging, or application diagnostics. That improves overall system efficiency and reduces disruption in the application lifecycles.

Op. scheduling is highly beneficial for long-running applications that perform multiple iterative computation cycles. A representative example demonstrating the benefits of *Op. scheduling* is the MHD application, which is granted thousands of compute nodes (each with 64 CPU cores and 8 GPUs). Its execution can last from days up to weeks, incurring significant computational costs in the LUMI system [2]. This application involves multiple computation cycles, with each cycle generating up to 2TB of data. Due to limited storage, new computation cycles must wait until data from previous cycles is processed, introducing inefficiencies. Thus, *Op. scheduling* is developed to leverage idle CPUs to process intermediate results, perform diagnostics, and move data before it is overwritten, improving resource utilization and reducing execution times.

B. Scheduling goals

The scheduler is designed with three objectives:

- Exploiting idle resources: The ODOP scheduler leverages ODOP monitoring to detect underutilized resources across multiple compute nodes. The detected idle resources will be allocated to *optasks* with minimal influences on the main application's performance. Given that provisioned resources are up to thousands of compute nodes, including a vast number of CPUs and GPUs for extended periods, optimizing resource utilization can lead to substantial cost savings.
- Minimizing time-to-solution: Scheduling *optasks* to run concurrently with the main application reduces time-to-solution by minimizing idle periods. By executing in parallel, *optasks* can complete necessary background computations, data preprocessing, or auxiliary operations, reducing idle time on the main workflow. This overlapping of computations reduces overall execution time compared to a strictly sequential execution. Thus, efficient scheduling of *optasks* enhances throughput and accelerates the time-to-solution.
- Customization according to application specific: ODOP scheduler should enable non-HPC experts (mostly domain scientists) to customize scheduling algorithms with minimal engineering effort. Many data-intensive applications require domain-specific knowledge to optimize task execution, data movement, and resource allocation effectively. Thus, the scheduler should provide an abstraction model that allows users to define/modify scheduling algorithms without requiring in-depth knowledge of the underlying infrastructures.

C. The differences between common scheduling and Op. scheduling

Op. scheduling differs from common task scheduling in several aspects. (1) Scheduling goals: Conventional schedulers

*As a technical report, which has not been in the peer review yet.

primarily allocate tasks on a certain amount of resources with multiple goals, e.g., load balancing, minimizing overheads, improving fairness, and task prioritization. Meanwhile, *Op. scheduling* execute *optasks* with minimal influence on the main tasks using underutilized resources, which are dynamic at runtime. (2) Resource management: Conventional schedulers in HPC operate at the system level, managed by the kernel or resource manager, while *Op. scheduling* is implemented at the application level by domain scientists, who have limited control over the allocated resources. (3) Resource requirements: *optasks* can execute on one or multiple computing nodes, with each node utilizing one or multiple cores, that can be specified by the domain scientists and depend on resource availability. (4) Dependency: *Op. scheduling* must account for data dependencies, as *optasks* are often triggered by intermediate outputs rather than fixed schedules. (5) Task execution patterns: *Op. scheduling* operates within long-running applications involving multiple computation cycles, so detecting resource utilization patterns allows *Op. scheduling* to improve task allocation and execution to reduce idle time and storage bottlenecks.

D. Challenges

Although scheduling algorithms have been extensively studied and developed to address various task scheduling problems, scheduling *optasks* across multiple computing nodes presents unique challenges that require careful examination.

- Dynamic resource availability: A challenge arises from the dynamicity of resource availability. In common scheduling problems, resource capacity is often fixed. However, *optasks* rely on idle resources, which fluctuate dynamically based on workload and the execution state of the main task. This variability complicates task provisioning, as resources may become unavailable before the allocated *optasks* complete. Meanwhile, saving checkpoints and resuming *optasks* at runtime can be computationally expensive. To ensure the completion of *optasks*, it is essential to estimate the availability window of resources. Otherwise, *optasks* may compete for resources with the main task, leading to contention and potential degradation of performance and increasing time-to-solution.
- *optask* diversity: Scheduling *optasks* is particularly challenging due to the diverse characteristics of these tasks, including varying resource requirements in terms of memory, CPU cores, and even the number of computing nodes. This problem shares similarities with bin packing, which is known to be NP-hard, but becomes even more difficult when diverse task dependencies and execution triggers are considered. Many *optasks* require specific data availability or periodic execution triggers, further constraining scheduling decisions
- Integration and execution overhead: Supporting domain scientists with multiple scheduling algorithms to accommodate diverse use cases while maintaining low overhead integration also presents a significant challenge. On the one hand, the scheduler should allow non-HPC experts to easily define or modify scheduling algorithms. That offers a variety of algorithms, potentially based on heuristics, to handle diverse task dependencies and resource availability. On the other hand, we must ensure the framework remains lightweight and incurs minimal execution overhead, which is critical to preventing performance degradation when deployed with the applications at runtime.

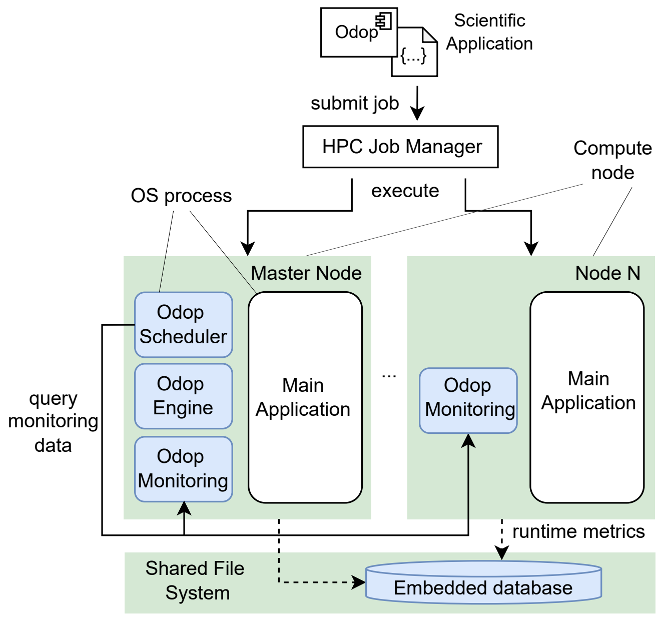

Fig. 1. An integration of Odop in an application managed by LUMI job submission (Slurm). Given multi-node execution, ODOP selects one of the compute nodes (as master node) to run Odop Scheduler and Odop Engine.

II. Design and Implementation

A. Overview

In most HPC systems, application execution is commonly managed through a job submission system (e.g., Slurm on LUMI [2]), so the ODOP scheduler is embedded within the application and submitted to a *HPC Job Manager*, as illustrated in Fig. 1. When the application runs on compute nodes, the main application will be executed normally, while the ODOP scheduler runs as a separate OS process, exclusively on the master node (e.g., the compute node with MPI rank 0). On the one hand, the ODOP scheduler continuously queries metrics from ODOP monitoring to estimate current available resources. On the other hand, the scheduler manages *optasks* via the ODOP engine deployed on the same master node.

B. Monitoring resource and execution task

1) Estimate resource availability: As mentioned in previous work [1], ODOP monitoring collects runtime metrics on the resource utilization of the application's processes and the compute node, storing them in an embedded database on a shared file system. The ODOP scheduler leverages APIs from ODOP monitoring to retrieve these metrics and estimate available resources. There will be one ODOP monitoring per compute node running the application, as illustrated in Fig. 2. To call these APIs from multiple compute nodes, the scheduler

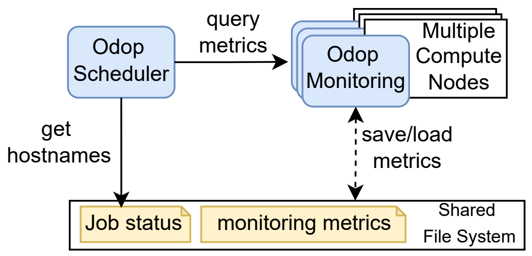

Fig. 2. Query metrics from multiple compute nodes

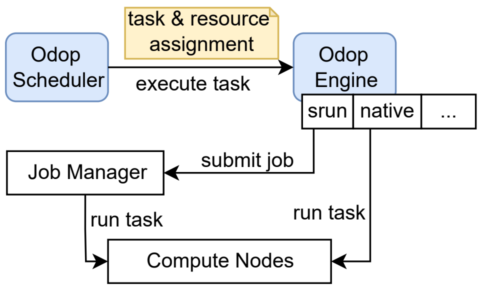

Fig. 3. Executing task

uses the hostnames (which correspond to the node IDs on the LUMI system). These hostnames are stored in a configuration file generated after the processes are launched.

Here, using REST APIs for querying metrics offers advantages in simplicity, lower overhead, minimal dependencies, and higher failure tolerance compared to MPI or direct process communication. REST APIs can provide an asynchronous and scalable solution, avoiding the complexity of managing MPI ranks, explicit synchronization, or manual socket handling, thus, more robust to failures. Although REST APIs rely on network communication, the network load for metrics is insignificant in HPC environments using InfiniBand with high-speed and low-latency connection.

2) Executing optasks*:* As shown in Fig. 3, the ODOP scheduler also utilizes REST APIs to communicate with the ODOP engine for managing the execution of *optasks*. Through this interaction, the scheduler can `start`, `stop`, and query *optasks* based on task ID, retrieving details such as status, process ID (PID), and task name. The commands of the scheduler will be interpreted into corresponding formats for specific underlying systems and execution methods (e.g., `srun` or native execution)

C. Op. scheduling

1) optask specification: provides essential information about the task so the scheduler can allocate suitable resources and execute the task at a specific period. As explained in previous work [1], *optasks* can be categorized based on characteristics such as resource requirements, deployment models (single-node or multi-node), and priority. To ensure that the execution of *optasks* does not influence the performance of the main application, our *optask* specification includes information as illustrated in Table I. For example, the specification includes

TABLE I
TASK SPECIFICATION

Attribute	**Description**
`function`	A callable function to execute the task
`name`	Task name
`priority`	Task priority
`time`	Estimated execution time
`memory`	Memory requirement
`nodes`	Number of computing node per task
`ranks_per_node`	Number of instances (replicas) per node
`cpus_per_rank`	Number of CPUs per instance
`disk_limit`	Disk requirement
`io_bound`	IO limitation (boolean)
`network_bound`	Network limitation (boolean)
`depends_on`	Task dependencies
`max_runs`	Max number of instances at the same time

the number of compute nodes needed, the number of instances (replicas) per node, the number of CPU cores per replica, memory requirements, priority level, and estimated execution time. By explicitly defining these parameters, the scheduler can make decisions on task placement on available resources.

2) Resource specifications: is used to illustrate the available resources. In most HPC systems, the compute nodes share the file systems, so the resource specification only consists of memory usage and CPU/GPU availability. To prevent *optasks* from interfering with each other when allocated on the same CPU/GPU, resource availability is represented as a boolean value (for a specific CPU core or GPU device) rather than a percentage number. CPU core availability is `True` when its utilization is below a configurable threshold (10% by default). Each CPU/GPU will be assigned to only one *optask*.

3) Scheduling algorithm : To minimize overhead, ODOP scheduler executes the *Op. scheduling* periodically at a configurable interval. The overall *Op. scheduling* workflow is illustrated in Fig. 4, providing an overview of how *optasks* are managed and executed. The *Op. scheduling* worflow consists of three main steps.

- **Step 1**: The scheduler selects tasks from the list of pending tasks while simultaneously querying monitoring data to estimate available resources across multiple compute nodes. The task selection algorithm can be flexible, depending on the use cases (e.g., priority or FIFO).
- **Step 2**: Based on the current available resources, the scheduler determines the most suitable compute nodes for execution. Resource allocation follows widely used algorithms such as best-fit, worst-fit, first-fit, and round-robin to ensure optimal task placement.
- **Step 3**: The scheduler assigns tasks by using various methods, including estimating resource availability duration or predicting task execution time using heuristics or ML-based forecasting techniques.

If a task is successfully assigned, it proceeds to task execution and repeats the three steps until no further tasks can be scheduled. Given that the *Op. scheduling* involves a combination of multiple algorithms within the three steps, their effectiveness must be evaluated across various scenarios through extensive

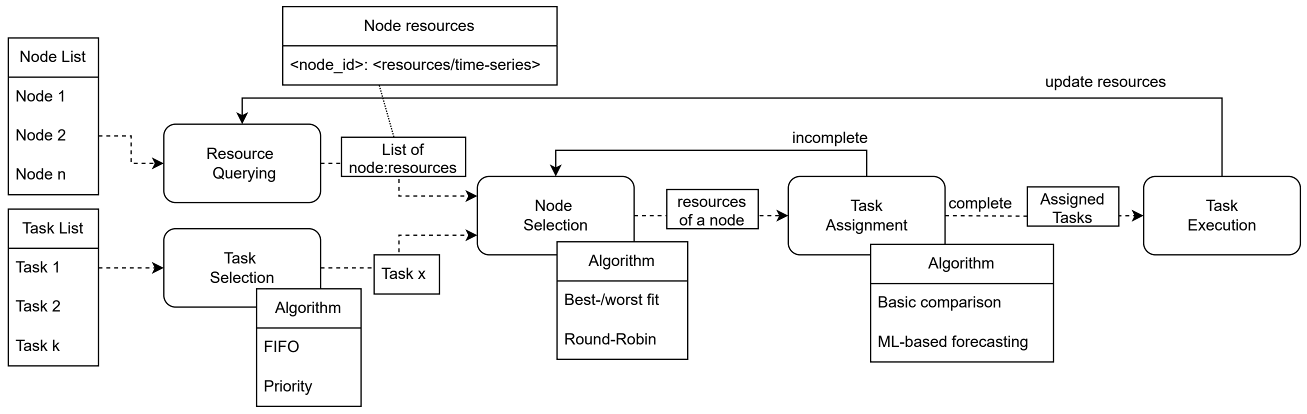

Fig. 4. Workflow within the scheduler

TABLE II
SCHEDULING ALGORITHMS

Algorithm	**Description**
FIFO	Select *optasks* for scheduling based on arrival time
Priority	Select *optasks* for scheduling based on arrival time with priority
Best-fit	Select resources for *optasks* with best-fit strategy
Round-robin	Select resources for *optasks* in round-robin manner
ML-based Estimation	An ML algorithm check if the *optasks* can be assigned to specific resources within specific time limits

experiments. This also highlights the importance of allowing domain scientists to select or implement their own scheduling algorithms with minimal engineering effort, ensuring adaptability to diverse applications and *optasks*.

4) Implementation: To enable domain scientists to customize algorithms for specific applications, we develop an abstract algorithm[1] that fully implements the three steps outlined in Section II-C3. As described, we provide resource and task specifications, allowing domain scientists to implement their own algorithms for scheduling *optasks* to the underutilized resources. New algorithms can be created by inheriting existing functions and modifying only the necessary functions according to specific optimization objectives. Algorithm selection is configured by specifying the algorithm name in the ODOP configuration, which maps to the corresponding implemented algorithm module. The implemented algorithms are listed in Table II.

Here, rule-based algorithms (e.g., FIFO, priority, and round-robin) require domain scientists to monitor resource usage and apply heuristics to optimize the scheduling in various scenarios. In contrast, ML-based algorithms can learn patterns but struggle to detect them across a large number of CPU cores and compute nodes. They often require online training since resource utilization patterns change based on application parameters or rely on complex models that introduce overhead, potentially affecting the main application process. Currently, we are developing an ML-based estimation algorithm to determine whether an optask can be assigned to specific resources within the expected time. This algorithm is still under development.

[1]https://github.com/rdsea/odop/tree/main/odop/scheduler/algorithms

III. Experiment

A. Experiment setup

In this section, we benchmark ODOP using two different applications: (1) GPT-2 Fine-Tuning – A large language model training application that processes massive datasets. The heavy computation is fully parallelized on the GPU, requiring only a minimal amount of CPU resources for orchestration. (2) MHD – An astrophysic simulation, currently implemented using Pencil-Code and Astaroth libraries to optimize intensive-computational tasks on GPUs. Thus, each process on a GPU device requires only a single CPU core for orchestrating computations. The remaining idle CPU cores can be leveraged for additional analysis tasks.

To evaluate ODOP, we use the following benchmarking criteria: (1) Total execution time – This includes the sequential execution time of both the main task and *optasks* (without ODOP) compared to when optasks are executed in parallel using underutilized resources (with ODOP). (2) Resource utilization – Measured as the average CPU and memory usage during execution. (3) Time-to-solution – The time required for domain scientists to make real-time decisions based on intermediate results. (4) Overhead of ODOP – The time to make decisions introduced by ODOP when scheduling and executing *optasks*.

B. Benchmarking results

1) Experiment with ML training: The GPT-2 fine-tuning application is currently implemented on a small scale due to the extensive training time required for processing massive datasets. At each checkpoint, the model can be exported and analyzed to help scientists adjust the training process. With ODOP, data processing and model upload can be executed concurrently with the training process by utilizing idle CPUs, without impacting the overall execution time of the main task (Table III).

TABLE III
BENCHMARK ODOP SCHEDULING WITH GPT-2 FINE-TUNING APPLICATION

No. GPUs	Optasks	Algorithm	Average CPU(%)	Execution time
2	No	No	3.52	1209.14
2	model-upload + data-processing	priority	5.02	1209.66
4	No	No	4.64	1024.07
4	model-upload + data-processing	priority	7.29	1017.37
8	No	No	9.22	789.73
8	model-upload + data-processing	priority	10.42	778.29

TABLE IV
OPTASK EXECUTION TIME

Optask	Nodes	Time(s)
PC-CPU	4	539
	32	757
	128	1089
Data-movement	4	73
	32	222
	128	947
Reduce	4	139.2

Currently, CPU utilization improvements in this application remain limited since we do not have many compute-intensive operations for the *optasks*. However, a significant portion of CPU resources remains underutilized, up to 90%.

2) Experiment with MHD application: For the MHD application, we conducted benchmarks at larger scales, utilizing 4, 32, and 128 compute nodes. We experimented with three optasks: (1) A secondary simulation run fully on the CPU (PC-CPU). (2) Single-node data reduction from snapshot data (reduce). (3) Uploading snapshot data from the simulation to cloud storage (data-movement).

In Table IV, we depict the execution time of each *optask* at different application scales. As shown in Tables V and VI, execution time improvements become more significant. In the PC-CPU case, ODOP reduces execution time by up to 51.61% comparing to sequential execution without ODOP. In addition, we measure the overhead of the ODOP scheduler during the task assignments in Table VII, which is significantly low and does not affect the performance of the application.

C. Customization

Domain scientists can select different algorithms when allocating *optasks*. As shown in Table VI, each algorithm offers varying optimization benefits in terms of execution time and average CPU usage. For instance, with 32 nodes and the PC-CPU *optask*, the round-robin algorithm results in the highest CPU utilization, whereas the best-fit algorithm achieves the shortest execution time. Since applications run with different configurations and the amount of data (to be processed in data operations) fluctuates significantly, extensive experimentation is required to help develop heuristics that assist domain scientists in selecting scheduling algorithms. Currently, due to the limited number of *optasks* in our experiments, the differences between algorithms are not yet clearly observable. In future work, we aim to support tasks with more diverse utilization patterns to provide deeper insights into optimization performance.

TABLE V
TASK SPECIFICATION

Numb node	Optasks	Average CPU (%)	Execution time (s)	Reduction (%)
4	No	17.79	666.75	0
	reduce+data-movement	18.38	728.75	48.6
	reduce	17.94	670.00	38.21
	PC-CPU	30.02	675.56	43.97
	data-movement	18.12	700.75	20.89
32	No	17.44	700.16	0
	reduce	17.3	719.6	-2.77 (failed)
	PC-CPU	29.92	704.98	51.61
	data-movement	16.8	934.455	31.6

TABLE VI
BENCHMARK ODOP SCHEDULING WITH MULTIPLE OPTASKS

No. nodes	Optasks	Algorithm	Average CPU(%)	Execution time
4	No	No	17.79	666.75
	reduce & data-movement	priority	18.70	709.00
		fifo	18.05	769.75
		best-fit	18.37	707.50
	reduce	round-robin	18.45	671.25
		priority	17.43	671.00
		fifo	17.89	667.00
		best-fit	18.00	671.00
	PC-CPU	round-robin	30.09	683.00
		priority	30.50	671.00
		fifo	29.82	673.00
		best-fit	30.40	675.25
	data-movement	round-robin	18.09	708.75
		priority	17.97	699.25
		fifo	18.30	694.25
32	No	No	17.44	700.16
	reduce	round-robin	17.52	719.47
		priority	17.21	728.94
		fifo	17.04	722.97
		best-fit	17.45	707.03
	PC-CPU	round-robin	29.99	710.91
		priority	29.86	711.06
		fifo	29.91	701.00
		best-fit	29.90	696.94
	data-movement	round-robin	16.72	948.88
		priority	17.44	777.00
		fifo	16.56	1007.00
		best-fit	16.47	1004.94
128	No	No	15.86	750.92
	PC-CPU	round-robin	28.56	773.57
		priority	28.65	743.45
		fifo	28.59	743.45
		best-fit	28.19	767.82
	data-movement	priority	14.64	1402.34
	data-movement	fifo	15.26	1119.33
	data-movement	best-fit	14.94	1361.25

D. Challenges and Limitations

Using ML algorithms to detect utilization patterns and predict resource availability duration presents several challenges due to its reliance on training with historical data.

TABLE VII
SCHEDULING OVERHEAD

No. node	Algorithm	Time(s)
4	Best-fit	0.0616
	FIFO	0.0635
	Round-robin	0.0871
	Priority	0.0627

However, scientists often modify application parameters between runs, leading to variations in utilization patterns. That makes offline training ineffective. On the other hand, online training introduces additional computational overhead, which can potentially influence other tasks.

The current challenge arises when integrating ODOP into an application that uses MPI (e.g., MHD), where multiple MPI processes are assigned to specific blocks of CPUs. Only one process (MPI rank 0) triggers the execution of the ODOP scheduler and engine in its sub-processes (independent OS process). Due to configuration and execution policy limitations in Slurm on the LUMI system, subprocesses cannot be allocated beyond the CPUs assigned to the parent process. As a result, the execution of *optasks* is significantly constrained in terms of allocation to specific CPUs, limiting flexibility and optimization in task distribution across the computing resources.

IV. Related work

Like most scheduling algorithms, *Op. scheduling* has to perform several decisions: task selection, resource selection, and task assignment. (1) Task Selection: various heuristics-based approaches exist. The most common heuristic algorithms are Min-Min and Max-Min [3], First come first serve (FCFS), Shortest job first (SFJ), Round Robin, Minimum completion time, and Suffrage [4]. In which, SJF seems to outperform in most common cases [5], [6], [7]. (2) Resource Selection: common strategies include best-fit, worst-fit, and first-fit, which determine optimal resource allocation based on availability. That can be applied for selecting computing nodes. (3) Task Assignment: simple rule-based approaches compare resource requirements with availability to make assignment decisions. Besides, machine learning (ML)-based methods such as He et al. [8] use the Markov chain model to predict if *optasks* can be allocated or not at a specific runtime.

Generalized scheduling can leverage meta-heuristic/AI algorithms such as swarm (Ant Colony [9], Bee Colony [10], Bacterial Foraging [11], Particle Swarm [12], Genetic Algorithm, and Differential Evolution [13]). However, these methods require extensive experimentation to generate sufficient data for heuristic decisions, making them computationally expensive and impractical for large-scale applications. Thus, these studies experiment with simulation tools, not the runtime of real applications.

V. Conclusions and Future Work

In this work, we implemented the ODOP scheduler with an abstract scheduling algorithm, which allows domain scientists to schedule *optasks* at the application level with minimal engineering effort. The experiment showed improvements in resource utilization and time-to-solution, thus potentially saving significant computation costs in HPC. Due to the lack of diversity in *optasks* for experiment and the difficulty in experimenting with applications in the development process, there are still massive resources underutilized. However, with the ODOP framework, we offer opportunities to customize scheduling algorithms that allow domain scientists to further improve their application workflows in the future.

Acknowledgment

This work has partially received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101118139. The authors wish to acknowledge the CSC IT Center for Science, Finland, for computing resources.

References

[1] M.-T. Nguyen, A.-D. Nguyen, J. Rantaharju, T. Puro, M. Rheinhardt, M. Korpi-Lagg, and H.-L. Truong, "Supporting opportunistic data operations for data-intensive computational applications," in *2024 IEEE International Conference on Big Data (BigData)*. IEEE, 2024, pp. 3735–3744.

[2] "Lumi supercomputer," https://lumi-supercomputer.eu/, accessed: 2024-09-18.

[3] S. H. H. Madni, M. S. Abd Latiff, M. Abdullahi, S. M. Abdulhamid, and M. J. Usman, "Performance comparison of heuristic algorithms for task scheduling in iaas cloud computing environment," *PloS one*, vol. 12, no. 5, p. e0176321, 2017.

[4] H. Krishnaveni and V. Sinthu Janita Prakash, "Execution time based sufferage algorithm for static task scheduling in cloud," in *Advances in big data and cloud computing: Proceedings of ICBDCC18*. Springer, 2019, pp. 61–70.

[5] S. Seth and N. Singh, "Dynamic heterogeneous shortest job first (dhsjf): a task scheduling approach for heterogeneous cloud computing systems," *International Journal of Information Technology*, vol. 11, no. 4, pp. 653–657, 2019.

[6] T. Nazar, N. Javaid, M. Waheed, A. Fatima, H. Bano, and N. Ahmed, "Modified shortest job first for load balancing in cloud-fog computing," in *Advances on Broadband and Wireless Computing, Communication and Applications: Proceedings of the 13th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA-2018)*. Springer, 2019, pp. 63–76.

[7] H. S. Caranto, W. C. L. Olivete, J. V. D. Fernandez, C. A. R. Cabiara, R. B. M. Baquirin, E. F. G. Bayani, and R. J. D. Fronda, "Integrating user-defined priority tasks in a shortest job first round robin (sjfrr) scheduling algorithm," in *Proceedings of 2020 6th International Conference on Computing and Data Engineering*, 2020, pp. 9–13.

[8] T. He, S. Chen, H. Kim, L. Tong, and K.-W. Lee, "Scheduling parallel tasks onto opportunistically available cloud resources," in *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE, 2012, pp. 180–187.

[9] S. Asghari and N. J. Navimipour, "Cloud service composition using an inverted ant colony optimisation algorithm," *International Journal of Bio-Inspired Computation*, vol. 13, no. 4, pp. 257–268, 2019.

[10] B. Hajimirzaei and N. J. Navimipour, "Intrusion detection for cloud computing using neural networks and artificial bee colony optimization algorithm," *Ict Express*, vol. 5, no. 1, pp. 56–59, 2019.

[11] F. Gao, F.-X. Fei, H.-q. Tong, and X.-j. Li, "Bacterial foraging optimization oriented by atomized feature cloud model strategy," in *Proceedings of the 32nd Chinese Control Conference*. IEEE, 2013, pp. 8032–8036.

[12] H. Ebrahimian, S. Barmayoon, M. Mohammadi, and N. Ghadimi, "The price prediction for the energy market based on a new method," *Economic research-Ekonomska istraživanja*, vol. 31, no. 1, pp. 313–337, 2018.

[13] M. Asafuddoula, T. Ray, and R. Sarker, "An adaptive hybrid differential evolution algorithm for single objective optimization," *Applied Mathematics and Computation*, vol. 231, pp. 601–618, 2014.