applied sciences

MDPI

Article

Efficient I/O Performance-Focused Scheduling in High-Performance Computing

Soeun Kim [1], Sunggon Kim [1] and Hwajung Kim [2,*]

[1] Department of Computer Science and Engineering, Seoul National University of Science and Technology, Seoul 01811, Republic of Korea; ybbsoen1230@seoultech.ac.kr (S.K.); sunggonkim@seoultech.ac.kr (S.K.)
[2] Department of Smart ICT Convergence Engineering, Seoul National University of Science and Technology, Seoul 01811, Republic of Korea
* Correspondence: hwajung.kim@seoultech.ac.kr

Abstract: High-performance computing (HPC) systems are becoming increasingly important as contemporary exascale applications with demand extensive computational and data processing capability. To optimize these systems, efficient scheduling of HPC applications is important. In particular, because I/O is a shared resource among applications and is becoming more important due to the emergence of big data, it is possible to improve performance by considering the architecture of HPC systems and scheduling jobs based on I/O resource requirements. In this paper, we propose a scheduling scheme that prioritizes HPC applications based on their I/O requirements. To accomplish this, our scheme analyzes the IOPS of scheduled applications by examining their execution history. Then, it schedules the applications at pre-configured intervals based on their expected IOPS to maximize the available IOPS across the entire system. Compared to the existing first-come first-served (FCFS) algorithm, experimental results using real-world HPC log data show that our scheme reduces total execution time by 305 h and decreases costs by USD 53 when scheduling 10,000 jobs utilizing public cloud resources.

Keywords: scheduling; I/O; high-performance computing; resource utilization; large-scale log; cloud

check for updates

Citation: Kim, S.; Kim, S.; Kim, H. Efficient I/O Performance-Focused Scheduling in High-Performance Computing. *Appl. Sci.* **2024**, *14*, 10043. https://doi.org/10.3390/app142110043

Academic Editor: Jose María Alvarez Rodríguez

Received: 10 October 2024
Revised: 29 October 2024
Accepted: 2 November 2024
Published: 4 November 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

As modern applications require more extensive computation and data processing capabilities, the role of large-scale distributed computing systems is becoming more important. In particular, high-performance computing (HPC) systems are becoming common execution environments for modern applications. In HPC systems, millions of active users run various applications, many of which require substantial computational resources. For example, applications in HPC systems often involve complex processes such as training and executing deep learning on large datasets. These tasks require extensive parallel processing across multiple nodes, and can generate numerous independent jobs within a single workflow. In such systems, a single application may be distributed across hundreds of nodes and thousands of computing devices, including central processing units (CPUs) and graphics processing units (GPUs), as well as across storage devices such as non-volatile memory express (NVMe) and serial advanced technology attachment (SATA) SSDs. Thus, efficient job scheduling is a complex yet essential task in HPC systems that plays a critical role in optimizing resource allocation while minimizing idle time and balancing system load [1]. Implementing efficient job scheduling can reduce user wait times and maximize overall throughput. This is accomplished by reducing the time that each job spends in a waiting queue, which enables more effective resource utilization. When resources are used to their fullest potential, HPC systems deliver more reliable performance, thereby enhancing user experience by providing faster and more consistent response times [2].

To provide efficient job scheduling, it is important to focus on factors that can cause bottlenecks and increase latency. One of the biggest contributors to performance bottlenecks in HPC systems is I/O performance [3]. The performance of many modern applications depends on efficient data access and storage operations. This includes database transactions, real-time analytics, and even compute-intensive tasks such as training machine learning models. For example, training even the basic BERT (Bidirectional Encoder Representations from Transformers) model [4] includes 110 million parameters. In addition, these models are often trained on huge datasets such as the Wikipedia corpus and BookCorpus, which total around 16 GB of uncompressed text data. During training, the model reads the data in mini-batches of 32,000 tokens for processing, and must also frequently write checkpoint files and intermediate results ranging from hundreds of megabytes to several gigabytes back to the disk. Thus, degraded I/O performance can significantly increase overall training times. For instance, a tenfold increase in read time can bottleneck the entire training process, leaving expensive computational resources such as GPUs and CPUs idle while waiting for data [5]. This inefficiency can ultimately extend training times from days to potentially weeks. Thus, even with powerful computational resources, inadequate I/O performance can become a critical limiting factor on overall system efficiency [6].

Therefore, in this paper we focus on the I/O characteristics of jobs to develop an efficient scheduling approach. We present an I/O-focused scheduling scheme for large-scale systems such as HPC implementations and cloud data centers. The goals of our scheme are (1) to decrease the total time required for executing jobs by implementing our scheduling scheme, and (2) to fully increase the utilization of HPC resources in order to maximize concurrent job execution. The key idea of our scheme is to prioritize the execution of jobs with the highest IOPS demands at designated time intervals in order to ensure maximum utilization of the available IOPS. To achieve this, our scheme first identifies the IOPS of different applications by analyzing their execution history. Then, by assessing the maximum available IOPS suitable for scheduling, the user sets the maximum IOPS for their system and defines an interval within which to check for executable applications. Finally, our scheme schedules those applications with the highest IOPS from the set of currently available applications while ensuring that the maximum IOPS limit is not exceeded.

We implemented and evaluated our scheme using large-scale log data obtained from real HPC systems, allowing us to capture diverse and unpredictable job patterns to demonstrate the robustness of our scheme. Moreover, our scheme is designed to be highly customizable, allowing for configuration adjustments based on the system characteristics and specific scheduling requirements of applications. We evaluated the performance of our proposed scheme in real-world scenarios while considering the various factors that different applications may encounter. The experimental results show that our scheme reduces the total execution time by 305 h, decreases costs by USD 53, and increases I/O utilization by 8% when scheduling 10,000 jobs utilizing public cloud resources.

Our contributions are as follows:

- We develop a scheme based on large-scale log data obtained from a real HPC system, ensuring robustness even for unpredictable and diverse job patterns.
- We design our scheme to be customizable, allowing it to be applied to different scheduling systems to meet varying requirements.
- We evaluate our proposed scheme in real-world scenarios, demonstrating its robustness and reliability for practical applications.

2. Background

2.1. Previous Studies on Scheduling for HPC

Job scheduling is crucial for enhancing the performance of HPC systems; consequently, many studies have focused on developing effective scheduling methods. However, traditional scheduling approaches often prioritize CPU and memory resources while neglecting the significant impact of I/O performance. Table 1 presents previous studies focusing on CPU scheduling strategies without considering I/O performance, while also emphasizing

the importance of I/O performance in scheduling. While these studies recognize that I/O performance must be considered along with CPU resources, they nevertheless focus on scheduling based on CPU requirements or I/O bandwidth without providing in-depth consideration of IOPS requirements. While the focus on improving the performance of HPC applications aligns with our objectives, the scheduling strategies in these papers differ significantly from ours. Our study specifically aims at job scheduling by prioritizing jobs according to their IOPS requirements rather than merely allocating I/O resources. Unlike from previous studies, our approach focuses on improving overall system performance by prioritizing jobs based on their I/O intensity.

Table 1. Previous scheduling studies.

Study	Target and Result	Focus
Siregar et al. [7]	Combines round-robin with a genetic algorithm for CPU scheduling, resulting in lower average waiting times than traditional methods.	CPU
Kadhim et al. [8]	Applies fuzzy logic in the design and implementation of a rule-based scheduling algorithm to solve the shortcoming of well-known scheduling algorithms (FCFS and SJF), achieving better waiting and turnaround times than priority scheduling with efficiency close to SJF while handling task priority.	CPU
Rajput et al. [9]	Analyzes a proposed algorithm with existing round-robin scheduling algorithm on the basis of varying time quantum, average waiting time, average turnaround time, and number of context switches, achieving reduced OS overhead and dispatch latency.	CPU
Carretero et al. [10]	Proposes a novel bandwidth-aware mapping algorithm for I/O list-scheduling policies, achieving improved system stretch and resource utilization while maintaining throughput. Evaluated strategies show better control of resource waste, especially with increased I/O nodes, optimizing performance for both throughput and latency.	I/O performance
Gainaru et al. [11]	Analyzes the effects of interference on application I/O bandwidth and proposes a scheduling technique to mitigate congestion, achieving reduced performance variability and improved system efficiency.	I/O performance
Zha et al. [12]	Proposes an adaptively periodic I/O scheduling (APIO) method by taking the stochastically iterative I/O periodicity of the applications into account, achieving improved system efficiency and reduced application dilation compared to existing methods.	I/O performance

2.2. Characteristics of Jobs in HPC Systems

HPC systems are designed to handle large-scale computation and data processing tasks that require substantial computing power and high IOPS [13]. Typical jobs performed on HPC systems include complex simulations, big data analytics, and AI model training and serving [14–16]. These workloads often include intensive I/O operations requiring quick data access, loading, and manipulation. In recent years there has been a significant increase in the demand for AI training and serving workloads on HPC systems. While AI models require compute-intensive workloads, efficient computation also requires careful consideration of I/O performance [17]. The machine learning and deep learning tasks central to AI models often involve I/O-intensive operations, making it essential to manage I/O effectively throughout the entire workflow. A notable example is the training of the GPT series models, such as GPT-3 and GPT-4, which requires extensive computational resources and high IOPS to handle the vast amounts of data involved [18,19]. These models

have billions of parameters, and need to rapidly access and update massive datasets during training. For instance, GPT-3, with 175 billion parameters, processes terabytes of text data, requiring the system to read and write data at high speeds to meet the computational demands of GPUs or TPUs [20]. Similarly, other deep learning tasks demand high IOPS for efficient data access and processing, especially during the training phase when large datasets are continuously processed. Ensuring high IOPS is crucial to prevent bottlenecks and optimize the utilization of computational resources [21].

2.3. Impact of IOPS on HPC Systems

The previous section establishes that jobs performed on HPC systems typically require high IOPS. The I/O performance of these jobs significantly impacts the overall system performance, making it a crucial factor in determining the efficiency of HPC systems. Ensuring adequate I/O performance is even more critical in these environments for two key reasons. First, as Figure 1 shows, the architecture of an HPC system uses a distributed file system, unlike computational resources such as CPUs and GPUs. As shown in the figure, App 1 (represented by the green box) uses three CPU nodes, each dedicated to separate functions, while App 2 (represented by the red box) utilizes four GPU nodes independently for its computations. However, both applications store their computation results in the shared OSS. This means that while the applications use different compute nodes (CPU and GPU) for computation-intensive workloads, the results and intermediate data are stored in shared storage (OSS), requiring all jobs to be allocated and scheduled for access to the DFS. This shared nature of DFS makes I/O performance susceptible to interference from multiple concurrently running applications, potentially causing contention and reduced performance. As noted by Kim et al. [5], concurrent access to DFS by multiple applications increases data access variability, leading to contention among applications. Similarly, Behzad et al. [6] highlight that this contention arises due to limited I/O bandwidth, which becomes overburdened by high-frequency read and write operations from various applications, resulting in resource bottlenecks and performance variability. Second, efficient I/O performance is essential for keeping all other resources fully utilized. Computation can only proceed if the input data are read from distributed storage and delivered to the node performing the computation in a timely manner. Delays in data transfer can result in data stalls, leaving nodes and their hardware idle and significantly degrading overall system efficiency [22,23]. Therefore, in order to schedule jobs without experiencing performance degradation it is crucial to focus on meeting the IOPS requirements of each job, thereby ensuring that sufficient I/O resources are allocated.

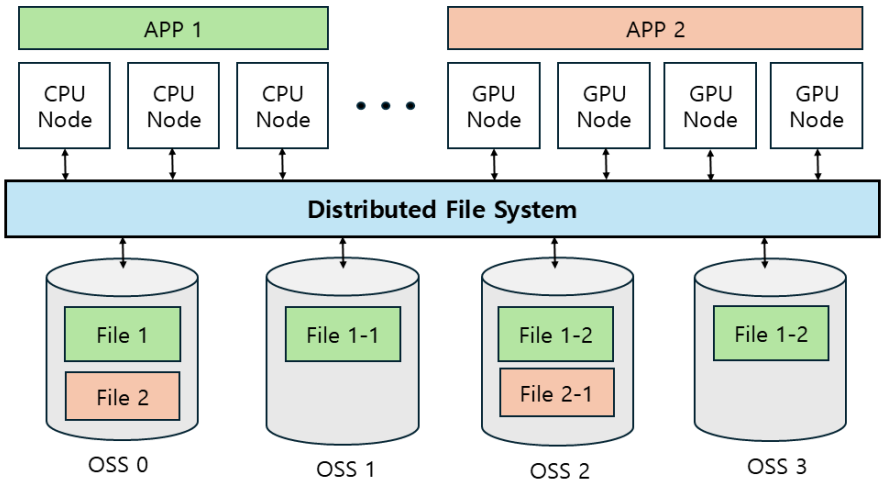


Figure 1. Architecture of HPC system with a distributed file system (OSS: Object Storage Server).

2.4. Prioritizing High IOPS Jobs

When scheduling jobs, it is crucial to correctly allocate the IOPS required by each job, as discussed in the previous section; in HPC systems, multiple jobs are requested simultaneously, making it inefficient to execute them one at a time [24]. Instead, as shown in Table 1, various strategies have been proposed to allocate these jobs for faster and more efficient scheduling. One such strategy is to prioritize those jobs that require the highest IOPS. By running these high-IOPS jobs first, the system can quickly complete tasks that need a lot of I/O resources, freeing up capacity for other jobs. High-IOPS jobs often need intense but short bursts of I/O, and when they are finished the remaining resources can be more easily allocated to jobs with lower IOPS needs. This approach ensures that high-IOPS tasks receive resources more quickly, thereby improving overall system efficiency by reducing I/O bottlenecks and enhancing resource management. By prioritizing high-IOPS tasks and delaying lower-priority jobs when I/O congestion is predicted, the system minimizes contention and reduces wait times for high-demand tasks [25]. Additionally, this prioritization helps to balance resource allocation, enabling smoother throughput and reducing delays from concurrent I/O-intensive workloads [26]. In contrast, if smaller jobs with lower IOPS requirements are scheduled first, they may occupy the I/O resources for a longer time, making it difficult to schedule high-IOPS jobs later. This can cause delays and lead to inefficient use of system resources, as high-IOPS jobs may have to wait in line for available I/O capacity. This inefficiency arises because high-IOPS jobs often rely on quickly accessing and processing large amounts of data to avoid bottlenecks in their execution. If these jobs are delayed, they can create a ripple effect that slows down the entire system. Moreover, high-IOPS jobs typically utilize a large portion of available I/O bandwidth for short periods [27,28]. If this bandwidth is occupied by smaller jobs, it can lead to resource fragmentation, where small chunks of I/O capacity are left unused but are insufficient to handle larger and more demanding jobs. This situation not only increases the overall waiting time but also prevents the system from fully utilizing its resources. Thus, scheduling high-IOPS jobs first ensures a smoother flow of tasks and minimizes idle times, thereby maximizing overall system efficiency.

3. Design

In this section, we present our scheme for effectively scheduling jobs in real-world HPC systems. The goals of the proposed scheme are as follows:

- **Customizability for Different Systems**: Users should be able to apply our scheme to any system by adjusting the Maximum IOPS and Check Time parameters (see Section 3.2).
- **Effective Job Scheduling**: Jobs should be scheduled with optimized resource allocation and minimal delay by selecting and executing the job with the highest IOPS that can be run at each Check Time (see Section 3.3).

3.1. Overall Procedure

Figure 2 shows the overall procedure of the proposed scheme. As shown in the figure, the overall procedure of our scheme consists of two components: Settings and Job Scheduling. First, we allow the user to set the max IOPS, which represents the highest IOPS that the system can offer. During this step, the user can also define a Check Time, which represents the interval at which the system checks for executable jobs. With this configuration, when job requests arrive they are schedule using four lists: waiting, working, excluded, and ended. At each Check Time, jobs are executed according to the states of these lists. By repeating this process, we ensure that all jobs are completed quickly, thereby improving the overall performance of the system.

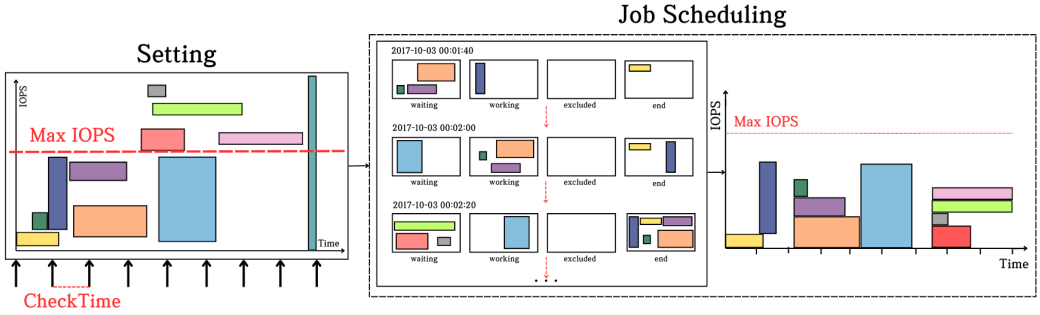


Figure 2. The overall procedure of our scheme: each box represents a job in the HPC application, with the x-axis representing time and the y-axis representing the IOPS of each job.

3.2. Defining Maximum IOPS and Check Time

In this step, we allow the user to define the maximum IOPS and check time parameters according to the performance of their specific system. The maximum IOPS represents the highest IOPS that the system can offer, and is determined by the storage performance. The check time represents the interval at which the system checks for executable jobs. In real-world scenarios, HPC systems have varying performance levels based on the devices that they use (e.g., CPU, storage). Therefore, to make our scheme adaptable for various HPC scheduling environments, it is crucial to establish appropriate criteria for the specific HPC system in use.

3.2.1. Set Maximum IOPS

If the maximum IOPS is high, scheduling becomes easier because almost every job can be executed simultaneously when needed. However, if the maximum IOPS is low, resources need to be used more efficiently, making scheduling more crucial in determining overall performance. Because HPC systems with high IOPS come with high storage costs, users cannot always select this setup [29]. Therefore, users may need to choose a more affordable system based on specific circumstances; as the environment changes, the scheduling logic should be adapted accordingly. In our scheme, users can configure the maximum IOPS to align with their specific requirements. Allowing dynamic adjustment of the maximum IOPS makes our scheme well-suited for dynamic systems, particularly in cloud environments. When scheduling on cloud systems, simply modifying the maximum IOPS parameter ensures adaptability across various virtual machine environments, such as AWS i4i and t4g instances [30]. Within our scheme, jobs are managed so as to avoid exceeding the set maximum IOPS while also maximizing I/O utilization. Therefore, to ensure scalability across diverse environments with minimal configuration changes and to maximize system utilization, our scheme must be designed to accommodate varying maximum IOPS levels across different HPC environments.

3.2.2. Set Check Time Interval

If the system checks for the best-fit job (i.e., the job with the highest IOPS) every time a new job arrives, this frequent checking process can introduce significant overhead, thereby reducing the overall performance of the system [31]. This issue is particularly critical in HPC environments, which often handle a massive volume of job requests. In such cases, setting an optimal check time interval becomes even more crucial. A too-frequent check time interval can waste computational resources on scheduling rather than job execution, while an interval that is too long may delay job execution and reduce system efficiency. Therefore, determining an appropriate check time interval is essential to balance the tradeoff between the overhead of frequent checking and the need for timely job execution. This approach ensures that our scheme remains efficient and adaptable across different HPC scenarios with varying job request volumes.

3.3. Scheduling

In this step, using the configured settings, we schedule jobs according to their IOPS. To achieve this, we create four lists to manage jobs based on their status.

3.3.1. Status Lists

Jobs do not arrive at regular intervals, and their arrival times often do not align with the check time. To ensure efficient scheduling, incoming jobs must be held until the next check time, when each job can be allocated the necessary resources for processing. This approach allows us to process jobs in batches, thereby minimizing the overhead associated with constantly checking for new arrivals. After a job begins processing, it is essential to monitor its status continuously in order to determine whether it has successfully completed or is still running. This ensures that resources are not wasted on jobs that have failed or stalled and that they can be reallocated to other pending tasks. Furthermore, jobs that are not suitable for the current HPC system due to insufficient resources or lower IOPS need to be promptly identified and excluded. By doing this, such jobs can be redirected to alternative environments that offer the required higher IOPS, preventing resource bottlenecks and improving overall system performance. To achieve this level of efficient scheduling, we categorize jobs into four statuses:

- **Waiting List:** Jobs that are waiting for the necessary resources to become available so they can begin execution.
- **Working List:** Jobs that are currently being executed using the assigned resources.
- **Excluded List:** Jobs that cannot be processed at the moment due to insufficient resources or other constraints.
- **Ended List:** Jobs that have successfully completed their execution.

When a job arrives, if its IOPS requirement does not exceed the maximum IOPS of the system, it is placed in the waiting list. If the IOPS of job requirement exceeds the maximum IOPS, then it is moved to the excluded list. Jobs that are currently allocated resources and are in the process of execution are placed in the Working list. After a job completes its execution, it is moved to the ended list. Based on the status of the jobs in the working and ended lists, we calculate the remaining IOPS that the system can offer. When a job starts execution and is added to the working list, we reduce the available IOPS by the amount required by that job, as those resources are now allocated; conversely, when a job completes and moves to the ended list, the IOPS initially assigned to it are released and added back to the pool of available IOPS, increasing the remaining IOPS for other jobs. Managing jobs based on these statuses can optimize resource allocation, reduce processing delays, and maintain high overall system efficiency.

3.3.2. Scheduling Method Based on IOPS Using Status Lists

At the specified check time, jobs from the waiting list are allocated utilizing the status lists, as explained above. In our scheme, all job requests except for those that exceed the maximum IOPS of the provided storage are placed in the waiting list, regardless of whether or not they can be executed immediately. Therefore, we only need to focus on the waiting list for scheduling. Jobs in the waiting list are sorted in descending order of IOPS; at check time, they are executed in that order until the maximum IOPS of the system is reached or there are no more jobs that can be allocated within the remaining IOPS. Executed jobs are then moved to the working list. When jobs in the working list are completed, they are immediately transferred to the ended list and the resources they used are released, increasing the available IOPS. In this way, the remaining resources available for use can be assessed at the next check time. By repeating this process at each check time until all job requests have been moved to the ended list, job scheduling can be managed efficiently.

3.3.3. Comparison of Job Scheduling Methods

As can be seen by examining the structural design plot of our scheme, scheduling jobs in this way is more efficient than using the basic scheduling model with the first-come first-served (FCFS) algorithm.

Figure 3 represents ten job requests that arrive in time order. In the figure, each box represents a job, the width of the box corresponds to the runtime of the job, and the

height corresponds to the required IOPS. By representing job requests as boxes, we can intuitively understand the time and resource allocation each job occupies. As can be seen from the figure, in this assumed environment there is no limitation on IOPS; thus, jobs can be executed immediately as they arrive. However, supporting such an environment is very expensive. To maximize resource utilization and optimize costs, scheduling is necessary. Figure 4 shows the scheduled job requests from Figure 3 using two methods: Graph A represents scheduling based on the FCFS approach, while Graph B represents scheduling based on our scheme. In the figure, each time interval represents a check time and the red dotted line indicates the maximum IOPS.

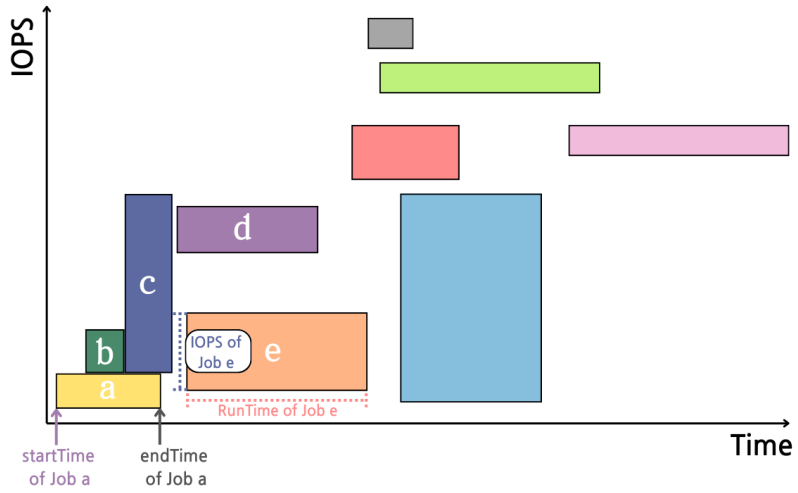


Figure 3. Without any IOPS limitations, jobs start working immediately upon arrival. Here, jobs are represented as boxes; the width corresponds to the runtime of the job, while the height represents the required IOPS. Thus, a wider box indicates a longer runtime, while a taller box signifies a job with high IOPS demands, requiring prompt assignment.

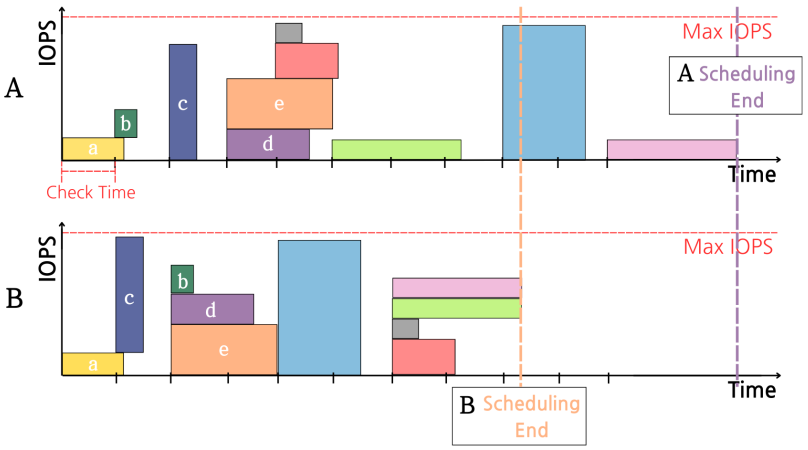


Figure 4. Here, jobs are scheduled with limitations on IOPS. Both graphs use the same jobs as those in Figure 3, with the original request times for each job also matching those in Figure 3. The upper graph shows the job execution simulation using the FCFS method, while the lower graph illustrates the simulation when using our scheme to prioritize jobs with higher IOPS requirements.

As shown in Figure 3, jobs (a,b,c,d,e) arrive in that order. In Graph A, jobs (a,b,c,d,e) are executed in the order they arrive at each check time. However, in Graph B they are executed in the order (a,c,e,d,b). Although jobs b and c both arrive after the first check time, job a is executed first in both graphs. At the second check time, the difference becomes apparent; in Graph B, job c is executed before job b because it has a higher IOPS, while in Graph A job b is executed first. This then affects the third check time; in Graph A,

jobs d and e cannot be executed due to insufficient remaining IOPS after job c. In Graph B, however, because job b, which has a lower IOPS, was executed earlier, enough IOPS remains to simultaneously execute jobs d and e. Therefore, the jobs are completed more quickly in Graph B, as demonstrated by the shorter end times shown in Figure 4. Even though this example uses only ten jobs, there is still a noticeable difference. As the number of jobs increases, the efficiency of our scheduling method becomes even more apparent.

4. Evaluation

4.1. Evaluation Setup

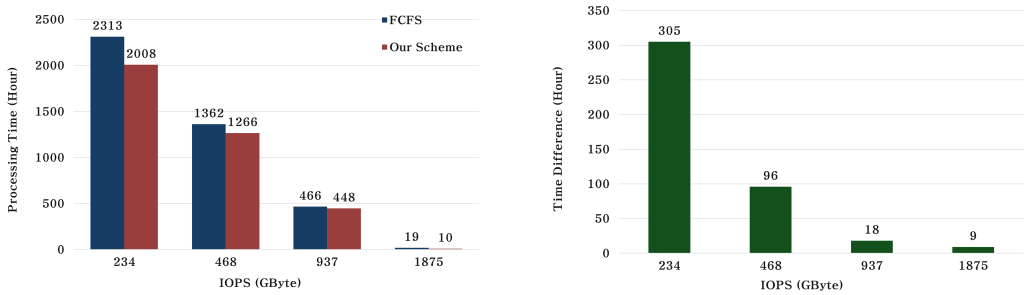
For the target system, we used logs from a real HPC system that operates a DFS with over 25,000 TB of capacity. The dataset comprised system logs that represent I/O behavior and resources for large-scale computational clusters, providing insights into I/O operations, bandwidth usage, and metadata activity, all of which care key factors for understanding and optimizing the performance of HPC applications. In large-scale systems, these logs are continuously collected to ensure correct operation, allowing us to analyze real-world applications. The dataset included logs spanning from 1 October 2017 to 31 January 2018. We processed these logs and generated graphs on a single compute node equipped with an AMD EPYC 7763 64-core CPU and four NVIDIA A100 GPUs. For software, we utilized Python 3.9.19 and TensorFlow 2.9.0. To test our scheme, we selected 10,000 logs in chronological order and set the check time to 20 s. We then compared our scheme with the FCFS algorithm, which is the standard method for scheduling jobs.

4.2. Evaluation Scenario

For evaluation, we set the check time to 20 s and conducted experiments by varying the maximum IOPS. By selecting 10,000 logs in chronological order, we aimed to simulate the scheduling process as realistically as possible. We used the start and end times to define the execution time of each job, simulating the scheduling process as if jobs were arriving in real time. When job requests arrived, they were marked as waiting jobs eligible for execution at any moment. At each check time, we executed the waiting jobs with the highest IOPS first, ensuring that the total IOPS did not exceed the set maximum IOPS limit. Additionally, if the required IOPS of a job exceeded the configured limit, we excluded it by adding it to an excluded list for further processing. When all 10,000 jobs were completed, we recorded the final end time as the scheduling end time.

4.3. Completion Time Difference

Effective scheduling ensures that all jobs are executed promptly, minimizing delays in delivering results to users. Therefore, we measured the completion time of all 10,000 jobs. Figure 5 illustrates the end time of all tasks at different IOPS levels. Figure 5a illustrates the specific time at which all jobs were completed. The x-axis represents the maximum IOPS, while the y-axis shows the completion time. As shown in Figure 5b, as the available IOPS increases, the gap in completion time between the FCFS algorithm and our scheme narrows. As the maximum IOPS increases, jobs can be allocated immediately without constraints, reducing the need for a scheduling scheme; however, in the smallest system configuration with a maximum IOPS of 234 GB, this difference amounts to 305 h. We only considered 10,000 jobs in this evaluation; however, in a real-world scenario the workload would be significantly larger, leading to an even greater difference in performance between the two scheduling methods. With higher IOPS availability, jobs can be processed more quickly upon arrival, reducing the need for complex scheduling based on IOPS requirements. This highlights the importance of an effective scheduling strategy when maximum IOPS is limited due to cost or availability constraints, especially in HPC systems. In practical scenarios, it is common to deal with lower IOPS capacity, making the choice of scheduling method even more crucial. Thus, our proposed scheme effectively reduces the overall completion time of jobs, particularly in environments with limited IOPS resources.

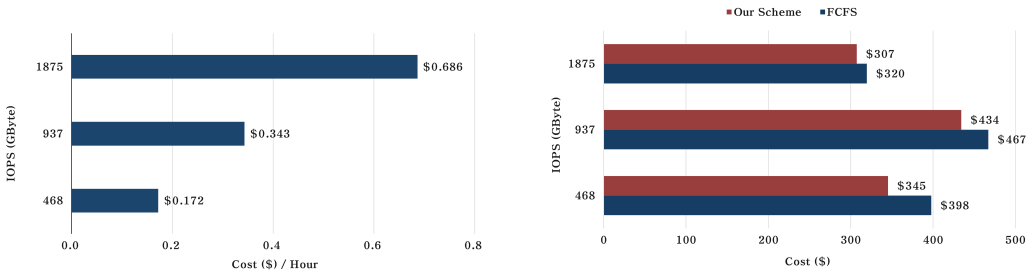


(**a**) (**b**)

Figure 5. Graphs representing the comparison of completion times. In (**a**), the completion times for the two cases (FCFS and our scheme) vary depending on the maximum IOPS; in (**b**), the difference in completion times (in minutes) between the two cases (FCFS and our scheme) varies depending on the maximum IOPS. In both graphs, the x-axis represents the configured maximum IOPS of the system. In (**a**), the y-axis indicates processing time in hours, while in (**b**) the y-axis shows the time difference in hours.

4.4. Cost Difference

One of primary purposes of scheduling is to utilize limited resources efficiently in order to reduce costs while optimizing performance. Therefore, comparing costs is a crucial aspect of this evaluation. In this study, we used the cost of AWS instances as a reference. AWS is one of the largest cloud providers and offers a variety of instances tailored for different use cases, including AI, machine learning, storage-optimized applications, and compute-optimized applications. The infrastructure supporting these requirements is essentially an HPC system that provides multiple compute nodes and high-performance storage. Figure 6a shows the cost of AWS i4i instances optimized for storage performance. In this figure, the x-axis indicates the hourly cost of each instance and the y-axis represents the maximum IOPS provided by each instance. As depicted, the cost increases with higher IOPS availability. In Section 4.3, we measured the completion time of 10,000 jobs. Based on this dataset, where the start time of the first job was 2017-10-01 1:48:42, we calculated the cost associated with different instances according to the provided IOPS. Figure 6b presents the cost information for executing 10,000 jobs on three different instances, illustrating the impact of IOPS on the total execution cost. As shown in the figure, the cost of using the FCFS algorithm is consistently higher than using our scheme across all IOPS levels. It is important to note that this evaluation is based on only 10,000 jobs. The cost difference would be even more pronounced in a real-world scenario with a significantly larger number of jobs, further highlighting the efficiency of our scheduling scheme.



(**a**) (**b**)

Figure 6. Graphs representing the comparison of cost: (**a**) the hourly cost per IOPS for optimized i4i instances offered by AWS and (**b**) the cost of running 10,000 jobs on i4i instances using the two different scheduling methods (FCFS and our scheme) at the available maximum IOPS for the i4i instances. In (**a**), the graph represents the system cost, which was assumed for evaluation purposes. We then used this cost to calculate the total expense required to execute all 10,000 jobs, with the results shown in (**b**).

4.5. I/O Throughput

Scheduling is essential for efficient resource utilization, as it enables us to determine the amount of resources such as IOPS that are needed at specific times. This information enables effective resource management and planning. When the resource requirements of a job exceed the available capacity, the system must take actions such as redistributing the workload across other resources by employing auto-scaling mechanisms. While such strategies ensure that the job is completed, they can also lead to additional costs if extra resources are used beyond what was initially allocated. Therefore, maximizing the utilization of existing resources is essential to avoid unnecessary expenses and maintain cost efficiency. By optimizing scheduling and effectively managing available resources, it is possible to prevent overprovisioning and reduce the need for costly scaling, ultimately resulting in better performance and cost savings. This opportunity underscores the importance of evaluating the effectiveness of different scheduling methods in resource utilization. Figure 7 shows the I/O throughput for each scheduling method. The x-axis represents the max IOPS, while the y-axis indicates the I/O throughput, calculated by averaging the I/O throughput across each check time. As illustrated in the figure, our scheduling scheme consistently achieves a higher resource utilization ratio compared to the FCFS algorithm across all maximum IOPS settings. The I/O throughput varies depending on how well the IOPS requirements of the jobs align with the maximum IOPS capacity. Therefore, it does not simply increase or decrease with higher maximum IOPS; rather, it depends largely on the method used to allocate jobs at each check time. By prioritizing jobs with higher IOPS requirements, we can achieve better resource utilization compared to scheduling jobs in the order of their arrival, as demonstrated in Section 3.3.3.

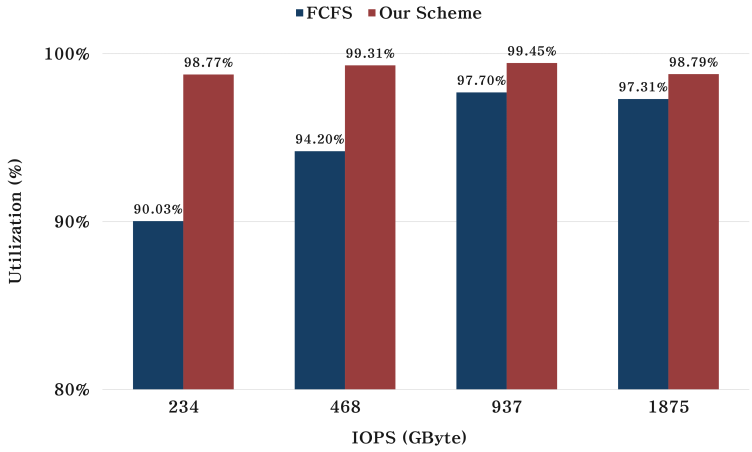


Figure 7. Graph showing the average I/O throughput at each check time for two scenarios measured at different maximum IOPS values. The x-axis represents the configured maximum IOPS, while the y-axis indicates the percentage of resource utilization.

4.6. Overhead

For practical use in real-time scenarios, it is essential to evaluate the overhead of our scheduling scheme. To analyze this, we measured the time required to schedule 10,000 jobs with a fixed check interval of 20 s under different maximum IOPS settings. With a maximum IOPS of 234 GB, the scheduling time was 362.33 s, while at 468 GB it was 306.31 s. As the maximum IOPS increased to 937 GB, the scheduling time decreased significantly to 161.63 s. Further increasing the maximum IOPS to 1875 GB reduced the scheduling time even more to 84.27 s. The variation in scheduling times across different maximum IOPS settings is due to the increased need for resource checks and allocations when the IOPS capacity is lower. With lower maximum IOPS, the system must perform more frequent checks and allocation operations over a longer period to efficiently manage all jobs. Most HPC systems provide more than 1000 GB of IOPS, enabling them to schedule at least 63 jobs per second, which is sufficient for real-time scheduling.

5. Related Work

5.1. CPU Scheduling

Numerous studies have sought to optimize the performance of HPC applications in large-scale systems through efficient scheduling techniques, with many focusing on CPU scheduling. For example, Siregar et al. [7] combined the round-robin method with a genetic algorithm to improve CPU scheduling, while Kadhim et al. [8] employed fuzzy logic to design and implement a rule-based scheduling algorithm to address the limitations of well-known algorithms such as FCFS and SJF. Similarly, Rajput et al. [9] analyzed a proposed scheduling algorithm in comparison to the traditional round-robin algorithm, considering factors such as varying time quantum, average waiting time, average turnaround time, and the number of context switches and demonstrating how their proposed method can overcome the shortcomings of round-robin scheduling for real-time applications. In contrast to these studies, our research focuses on I/O for job scheduling rather than on CPU.

5.2. I/O Scheduling

Similar to our focus, several previous studies have aimed to improve the performance of HPC applications through effective I/O scheduling. For example, Carretero et al. [10] proposed a bandwidth-aware mapping algorithm for I/O scheduling, while Gainaru et al. [11] investigated the impact of interference on application I/O bandwidth and suggested a scheduling technique to alleviate congestion. Zha et al. [12] developed an adaptively periodic I/O scheduling (APIO) method that considers the stochastically iterative I/O periodicity of applications. In our study, however, we focus on IOPS as a key metric for scheduling decisions. Instead of concentrating on managing I/O resources, we design an I/O scheduling algorithm that prioritizes jobs with higher IOPS requirements, ensuring that tasks with high I/O demand are scheduled first.

6. Future Work

While this study successfully introduces a new scheme that can schedule jobs based on IOPS, there are still several aspects that require further exploration to make this scheme more practical. Although we simulated job scheduling using the IOPS levels typically offered by AWS, we were unable to fully integrate our scheme into real-world scheduling algorithms. This gap suggests that our scheme, while promising, has not yet been tested against the dynamic and complex requirements of actual cloud environments. To realize the full potential of this scheme, it is crucial to incorporate it into existing scheduling frameworks such as those used by AWS. Real-world scheduling involves a range of factors that can significantly influence scheduling decisions, such as varying workloads, resource availability, and system performance fluctuations. By adapting our scheme to these real-world scenarios, it could be refined and strengthened in terms of both robustness and effectiveness. Addressing these remaining challenges would help to validate our scheme under realistic conditions as well as to enhance its applicability and impact in practical cloud operations.

7. Conclusions

In this paper, we have proposed a scheduling scheme for HPC jobs based on high IOPS demands. Through the presented approach, we developed a scheduling mechanism that effectively achieves our primary goals, namely, reducing total execution time and increasing resource utilization. First, our evaluation shows that our scheme reduces the total execution time for 10,000 jobs with a maximum IOPS of 234 GB by 305 h compared to the FCFS algorithm, aligning with our goal of decreasing overall execution time. Second, our scheme achieves a resource utilization rate of 98.77% under the same IOPS conditions, representing an 8% improvement over FCFS and meeting our objective of maximizing resource utilization. Moreover, in addition to saving time and resources, our scheme's emphasis on flexibility sets it apart from other algorithms. Unlike traditional approaches, we have focused on creating a model that is adaptable to various HPC systems. To enable

this, our scheme allows users to configure the check time interval and maximum IOPS parameters according to their specific requirements, making it both adaptable and robust across diverse HPC environments. This flexibility ensures that our scheme can effectively manage varying workloads and dynamically changing conditions, offering a significant boost in scheduling efficiency and performance. By prioritizing high-IOPS jobs in this way, our scheme successfully enhances scheduling performance in terms of both time and resource efficiency while also increasing adaptability for a range of applications.

Author Contributions: Conceptualization, S.K. (Soeun Kim) and S.K. (Sunggon Kim); software, S.K. (Soeun Kim) and S.K. (Sunggon Kim); visualization, S.K. (Soeun Kim); formal analysis, S.K. (Soeun Kim) and S.K. (Sunggon Kim); writing—original draft preparation, S.K. (Soeun Kim); writing—review and editing, S.K. (Sunggon Kim) and H.K.; project administration, S.K. (Soeun Kim); funding acquisition, H.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. RS-2023-00240734).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data is contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Paul, A.K.; Goyal, A.; Wang, F.; Oral, S.; Butt, A.R.; Brim, M.J.; Srinivasa, S.B. I/o load balancing for big data hpc applications. In Proceedings of the 2017 IEEE International Conference on Big Data (Big Data), Boston, MA, USA, 11–14 December 2017; pp. 233–242.
2. Tawfeek, M.A.; El-Sisi, A.; Keshk, A.E.; Torkey, F.A. Cloud task scheduling based on ant colony optimization. In Proceedings of the 2013 8th International Conference on Computer Engineering & Systems (ICCES), Cairo, Egypt, 26–28 November 2013; pp. 64–69.
3. Luu, H.; Behzad, B.; Aydt, R.; Winslett, M. A multi-level approach for understanding I/O activity in HPC applications. In Proceedings of the 2013 IEEE International Conference on Cluster Computing (CLUSTER), Indianapolis, IN, USA, 23–27 September 2013; pp. 1–5.
4. Koroteev, M.V. BERT: A review of applications in natural language processing and understanding. *arXiv* **2021**, arXiv:2103.11943.
5. Kim, S.; Sim, A.; Wu, K.; Byna, S.; Son, Y. Design and implementation of I/O performance prediction scheme on HPC systems through large-scale log analysis. *J. Big Data* **2023**, *10*, 65. [CrossRef]
6. Behzad, B.; Byna, S.; Prabhat; Snir, M. Optimizing i/o performance of hpc applications with autotuning. *ACM Trans. Parallel Comput. (TOPC)* **2019**, *5*, 1–27. [CrossRef]
7. Siregar, M.U. A new approach to CPU scheduling algorithm: Genetic round robin. *Int. J. Comput. Appl.* **2012**, *47*, 18–25.
8. Kadhim, S.J.; Al-Aubidy, K.M. Design and evaluation of a fuzzy-based CPU scheduling algorithm. In *International Conference on Business Administration and Information Processing*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 45–52.
9. Rajput, I.S.; Gupta, D. A priority based round robin CPU scheduling algorithm for real time systems. *Int. J. Innov. Eng. Technol.* **2012**, *1*, 1–11.
10. Carretero, J.; Jeannot, E.; Pallez, G.; Singh, D.E.; Vidal, N. Mapping and scheduling HPC applications for optimizing I/O. In Proceedings of the 34th ACM International Conference on Supercomputing, Barcelona, Spain, 29 June–2 July 2020; pp. 1–12.
11. Gainaru, A.; Aupy, G.; Benoit, A.; Cappello, F.; Robert, Y.; Snir, M. Scheduling the I/O of HPC applications under congestion. In Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium, Hyderabad, India, 25–29 May 2015; pp. 1013–1022.
12. Zha, B.; Shen, H. Adaptively periodic I/O scheduling for concurrent HPC applications. *Electronics* **2022**, *11*, 1318. [CrossRef]
13. Choi, J.; Dukhan, M.; Liu, X.; Vuduc, R. Algorithmic time, energy, and power on candidate HPC compute building blocks. In Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, 19–23 May 2014; pp. 447–457.
14. Liao, C.; Lin, P.H.; Verma, G.; Vanderbruggen, T.; Emani, M.; Nan, Z.; Shen, X. Hpc ontology: Towards a unified ontology for managing training datasets and ai models for high-performance computing. In Proceedings of the 2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC), St. Louis, MO, USA, 15 November 2021; pp. 69–80.
15. Sukumar, S.R.; Balma, J.A.; Rickett, C.D.; Maschhoff, K.J.; Landman, J.; Yates, C.R.; Chittiboyina, A.G.; Peterson, Y.K.; Vose, A.; Byler, K.; et al. The convergence of HPC, ai and Big Data in rapid-response to the COVID-19 pandemic. In *Smoky Mountains Computational Sciences and Engineering Conference*; Springer: Cham, Switzerland, 2021; pp. 157–172.

16. Jiang, Z.; Gao, W.; Wang, L.; Xiong, X.; Zhang, Y.; Wen, X.; Luo, C.; Ye, H.; Lu, X.; Zhang, Y.; et al. HPC AI500: A benchmark suite for HPC AI systems. In *Benchmarking, Measuring, and Optimizing: First BenchCouncil International Symposium, Bench 2018, Seattle, WA, USA, 10–13 December 2018, Revised Selected Papers 1*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 10–22.
17. Jia, X.; Zhang, Y.; Liu, G.; Yang, X.; Zhang, T.; Zheng, J.; Xu, D.; Liu, Z.; Liu, M.; Yan, X.; et al. XVDPU: A High-Performance CNN Accelerator on the Versal Platform Powered by the AI Engine. *ACM Trans. Reconfig. Technol. Syst.* **2024**, *17*, 1–24. [CrossRef]
18. Chen, L.; Lin, P.H.; Vanderbruggen, T.; Liao, C.; Emani, M.; De Supinski, B. Lm4hpc: Towards effective language model application in high-performance computing. In *International Workshop on OpenMP*; Springer: Cham, Switzerland, 2023; pp. 18–33.
19. Muhamed, A.; Bock, C.; Solanki, R.; Park, Y.; Wang, Y.; Huan, J. Training Large-scale Foundation Models on Emerging AI Chips. In Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Long Beach, CA, USA, 6–10 August 2023; pp. 5821–5822.
20. Brown, T.B. Language models are few-shot learners. *arXiv* **2020**, arXiv:2005.14165.
21. Kindratenko, V.; Mu, D.; Zhan, Y.; Maloney, J.; Hashemi, S.H.; Rabe, B.; Xu, K.; Campbell, R.; Peng, J.; Gropp, W. Hal: Computer system for scalable deep learning. In Proceedings of the Practice and Experience in Advanced Research Computing, Portland, OR, USA, 26–30 July 2020; pp. 41–48.
22. Imamura, S.; Yoshida, E. Reducing cpu power consumption for low-latency ssds. In Proceedings of the 2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA), Hakodate, Japan, 28–31 August 2018; pp. 79–84.
23. Soundararajan, G.; Amza, C. Towards end-to-end quality of service: Controlling I/O interference in shared storage servers. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 287–305.
24. Herbein, S.; Ahn, D.H.; Lipari, D.; Scogland, T.R.; Stearman, M.; Grondona, M.; Garlick, J.; Springmeyer, B.; Taufer, M. Scalable I/O-aware job scheduling for burst buffer enabled HPC clusters. In Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, Kyoto, Japan, 31 May–4 June 2016; pp. 69–80.
25. Wu, K.; Wei, J.; Lin, J. SchedP: I/O-aware Job Scheduling in Large-Scale Production HPC Systems. In *IFIP International Conference on Network and Parallel Computing*; Springer: Cham, Switzerland, 2022; pp. 315–326.
26. Zhou, Z.; Yang, X.; Zhao, D.; Rich, P.; Tang, W.; Wang, J.; Lan, Z. I/O-aware batch scheduling for petascale computing systems. In Proceedings of the 2015 IEEE International Conference on Cluster Computing, Chicago, IL, USA, 8–11 September 2015; pp. 254–263.
27. Fan, Y.; Lan, Z.; Rich, P.; Allcock, W.; Papka, M.E. Hybrid workload scheduling on HPC systems. In Proceedings of the 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Lyon, France, 30 May–3 June 2022; pp. 470–480.
28. Kim, Y.; Gunasekaran, R. Understanding I/O workload characteristics of a Peta-scale storage system. *J. Supercomput.* **2015**, *71*, 761–780. [CrossRef]
29. Munhoz, V.; Castro, M.; Mendizabal, O. Strategies for fault-tolerant tightly-coupled hpc workloads running on low-budget spot cloud infrastructures. In Proceedings of the 2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Bordeaux, France, 2–5 November 2022; pp. 263–272.
30. Wittig, A.; Wittig, M. *Amazon Web Services in Action: An In-Depth Guide to AWS*; Simon and Schuster: New York, NY, USA, 2023.
31. Lyakhovets, D.; Baranov, A. Efficiency thresholds of group based job scheduling in HPC systems. *Lobachevskii J. Math.* **2022**, *43*, 2863–2876. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.